SCO UNIX® System V/386

Development System

C Language Reference

The Santa Cruz Operation, Inc.

Portions © 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989 Microsoft Corporation.
All rights reserved.
Portions © 1989 AT&T.
All rights reserved.
Portions © 1983, 1984, 1985, 1986, 1987, 1988, 1989 The Santa Cruz Operation, Inc.
All rights reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal, Santa Cruz, California, 95062, U.S.A. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

The copyrighted software that accompanies this manual is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

USE, DUPLICATION, OR DISCLOSURE BY THE UNITED STATES GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH (c) (1) OF THE COMMERCIAL COMPUTER SOFTWARE -- RESTRICTED RIGHTS CLAUSE AT FAR 52.227-19 OR SUBPARAGRAPH (c) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND COMPUTER SOFTWARE CLAUSE AT DFARS 52.227-7013. ''CONTRACTOR/ MANUFACTURER'' IS THE SANTA CRUZ OPERATION, INC., 400 ENCINAL STREET, P.O. BOX 1900, SANTA CRUZ, CALIFORNIA, 95061, U.S.A.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.
Intel is a registered trademark of Intel Corporation.
UNIX is a registered trademark of AT&T.

SCO Document Number: 6-26-89-6.0/3.2.0

Contents

1 **Introduction**

Overview of the C Language 1-1
About This Manual 1-3
Notational Conventions 1-5

2 **Elements of C**

Introduction 2-1
Character Sets 2-2
Constants 2-10
Identifiers 2-17
Keywords 2-19
Comments 2-20
Tokens 2-22

3 **Program Structure**

Introduction 3-1
Source Program 3-2

4 **Declarations**

Introduction 4-1
Type Specifiers 4-2
Declarators 4-9
Variable Declarations 4-17

5 **Expressions and Assignments**

Introduction 5-1
C Operators 5-14
Assignment Operators 5-32
Precedence and Order of Evaluation 5-37
Type Conversions 5-41

6 **Statements**

Introduction 6-1
The break Statement 6-3
The Compound Statement 6-4
The continue Statement 6-5
The do Statement 6-6

The Expression Statement 6-7
The for Statement 6-9
The goto and Labeled Statements 6-11
The if Statement 6-13
The Null Statement 6-15
The return Statement 6-16
The switch Statement 6-18

7 Functions

Introduction 7-1
Function Definitions 7-3

8 Preprocessor Directives and Pragmas

Introduction 8-1
Manifest Constants and Macros 8-3
Include Files 8-12
Conditional Compilation 8-14
Line Control 8-19
Pragmas 8-21

A Differences Between K&R C and Microsoft C

Introduction A-1

B Syntax Summary

Tokens B-1
Expressions B-7
Declarations B-9
Statements B-14
Definitions B-15
Preprocessor Directives B-16
Pragmas B-17

Chapter 1

Introduction

Overview of the C Language 1-1

About This Manual 1-3

Notational Conventions 1-5

Overview of the C Language

The C language is a general-purpose programming language known for its efficiency, economy, and portability. While these characteristics make it a good choice for almost any kind of programming, C has proven especially useful in systems programming because it facilitates writing fast, compact programs that are readily adaptable to other systems. Well-written C programs are often as fast as assembly-language programs, and they are typically easier for programmers to read and maintain.

C was designed to combine efficiency and power in a relatively small language. C does not include built-in functions to perform tasks such as input and output, storage allocation, screen manipulation, and process control. To perform such tasks, C programmers rely on ‘‘run-time libraries,’’ a set of predefined functions and macros. The run-time library functions available for use in Microsoft® C programs are discussed in a separate manual, the *UNIX System V C Library Guide*.

C’s design makes it both flexible and compact. Because the language is relatively sparse, it neither assumes nor imposes a particular programming model. You can use the run-time routines supplied, or tailor your own variations for special purposes. The design also helps to isolate language features from processor-specific features in a particular C implementation, which makes it easier to write portable code. While the strict definition of the language makes it independent of any particular operating system or machine, you can easily add system-specific routines to take advantage of the most efficient features of a particular machine.

Note

> Microsoft is committed to conformity with the developing standard for the C language as set forth in the Draft Proposed American National Standard — Programming Language C (hereinafter referred to as the ANSI C standard. Microsoft extensions to the ANSI C standard are noted in the text. Because the extensions are not a part of the ANSI C standard, their use may restrict portability of programs between systems. See your compiler guide for information on enabling and disabling Microsoft extensions.

The C language includes the following significant features:

- A full set of loop, conditional, and transfer statements to control program flow logically and efficiently and to encourage structured programming.
- A large set of operators. Many of these operators correspond to common machine instructions, allowing a direct translation into machine code. The variety of operators allows you to specify different kinds of operations clearly and with a minimum of code.
- Several sizes of integers, as well as single- and double-precision floating-point types. You can also design more complex data types, such as arrays and data structures, to suit specific program needs.
- Declarations of ''pointers'' to variables and functions. A pointer to an item corresponds to the item's machine address. Pointers can make programs more efficient, since they let you refer to items in the same way the machine does. C also supports pointer arithmetic, which lets you access and manipulate memory addresses directly.
- A C preprocessor that acts on the text of files before they are compiled. You can use the C preprocessor to define program constants, substitute fast macro definitions for function calls, and compile parts of programs based on specified conditions.

C is a flexible language that leaves many programming decisions up to you. In keeping with this philosophy, C imposes few restrictions in matters such as type conversion. Although this characteristic of the language can make your programming job easier, you must know the language well to understand how programs will behave.

About This Manual

The *C Language Reference* defines the C language as implemented by Microsoft Corporation. It is intended as a reference for programmers experienced in C or other programming languages. Thorough knowledge of programming fundamentals is assumed.

Consult your compiler guide for an explanation of how to compile and link C programs on your system; this manual also contains information specific to the implementation of C on your system.

This manual is organized as follows:

Chapter 1, ''Introduction,'' introduces this guide and outlines the notational conventions used in this manual.

Chapter 2, ''Elements of C,'' describes the letters, numbers, and symbols that can be used in C programs and the combinations of characters that have special meanings to the C compiler.

Chapter 3, ''Program Structure,'' discusses the components and structure of C programs and explains how C source files are organized.

Chapter 4, ''Declarations,'' describes how to specify the attributes of C variables, functions, and user-defined types. C provides a number of predefined data types and lets the programmer declare ''aggregate'' types and pointers. Function prototypes, a relatively new feature of C, are discussed in this chapter, as well as in Chapter 7, ''Functions.''

Chapter 5,''Expressions and Assignments,'' describes the operands and operators that form C expressions and assignments. The chapter also discusses the type conversions and side effects that may occur when expressions are evaluated.

Chapter 6, ''Statements,'' describes C statements, which control the flow of program execution.

Chapter 7, ''Functions,'' discusses C functions. In particular, this chapter explains function prototypes, formal parameters, and return values. It also describes how to define, declare, and call functions.

Chapter 8, ''Preprocessor Directives and Pragmas,'' describes the instructions recognized by the C preprocessor, a text processor that is automatically invoked before compilation. This chapter also introduces ''pragmas,'' special instructions to the compiler that you can place in source files.

Appendix A, ''Differences,'' lists the differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

Appendix B, ''Syntax Summary,'' summarizes the syntax of the C language as implemented by Microsoft.

Notational Conventions

This manual uses the following notational conventions:

keywords	Bold type indicates text that must be typed exactly as shown. Text that is shown in bold type includes C keywords, such as **goto** and **char**, and operators, such as the addition operator (**+**) and the multiplication operator (*****).
placeholders	Terms in italics may appear in syntax descriptions or in the text. In these instances, the terms are being used as placeholders that you would replace with specific terms or values in an actual C program. For example, in **goto** *name***;** *name* appears in italics to show that this is a general form for the **goto** statement. In an actual program statement, you must supply a particular identifier for the placeholder *name*. Occasionally, italics are used to emphasize particular words in the text.
`Examples`	Examples of C programs and program elements appear in a special typeface to look similar to listings on the screen or the output of commonly used computer printers:

```
int x, y;
.
.
.
swap (&x, &y);
```

Input: *output*	Some examples show both program output and user input; in these cases, input is shown in a darker font.

Repeating . . . elements

Vertical ellipsis dots are used in program examples or syntax to indicate that a portion of the program is omitted.

In the following example, the vertical ellipsis dots indicate that zero or more declarations, followed by one or more statements, may appear between the braces:

{
[*declaration*]
.
.
.
statement
[*statement*]
.
.
.
}

In the following excerpt, two program lines are shown. The ellipsis dots between the lines indicate that additional program lines appear between these two lines but are not shown:

```
int x, y;
.
.
.
swap (&x, &y);
```

Horizontal ellipsis dots following an item indicate that more items of the same form may appear. For instance,

= {*expression* [, *expression*]...}

indicates that one or more expressions separated by commas may appear between the braces ({ }).

[*optional items*]	Brackets enclose optional items in syntax descriptions. For example, **return** [*expression*]; is a syntax description showing that *expression* is an optional item in the **return** statement. Single brackets are used to indicate brackets used by C-language array declarations and subscript expressions. For instance, *a[10]* is an example of brackets in a C subscript expression.
‘‘Defined terms’’	Quotation marks set off terms defined in the text. For example, the term ‘‘token’’ appears in quotation marks when it is defined. Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form " " rather than ‘‘ ’’. The following example shows a C string: `"abc"` Quotation marks also occasionally indicate a term that is being used in a colloquial sense.
KEY+NAMES	Names of special key combinations, such as CTRL+Z, appear in small capital letters.

Chapter 2

Elements of C

Introduction 2-1

Character Sets 2-2
  Letters, Digits, and Underscore 2-3
  White-Space Characters 2-3
  Punctuation and Special Characters 2-4
  Escape Sequences 2-5
  Operators 2-7

Constants 2-10
  Integer Constants 2-10
  Floating-Point Constants 2-12
  Character Constants 2-13
  String Literals 2-14

Identifiers 2-17

Keywords 2-19

Comments 2-20

Tokens 2-22

Introduction

This chapter describes the elements of the C programming language, including the names, numbers, and characters used to construct a C program. The following topics are discussed in this chapter:

- Character sets
- Constants
- Identifiers
- Keywords
- Comments
- Tokens

Character Sets

Two character sets are defined for use in C programs: the ‘‘C character set’’ and the ‘‘representable character set.’’

The C character set consists of the letters, digits, and punctuation marks having specific meanings in the C language. You construct a C program by combining the characters of the C character set into meaningful statements.

The C character set is a subset of the representable character set. The representable character set includes each letter, digit, and symbol that can be represented graphically with a single character. The extent of the representable character set depends on the type of terminal, console, or character device being used.

All characters in a C program must be part of the C character set. However, string literals, character constants, comments, and file names in **#include** directives can include any character from the representable character set.

Since each character in the C character set has an explicit meaning in the language, the compiler generates error messages when it finds inappropriate or inappropriately used characters in a program.

The sections that follow describe the characters and symbols of the C character set and explain how and when to use them.

Letters, Digits, and Underscore

The C character set includes the uppercase and lowercase letters of the English alphabet, the 10 decimal digits of the Arabic number system, and the underscore (_) character.

- Uppercase English letters

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Lowercase English letters

  a b c d e f g h i j k l m n o p q r s t u v w x y z

- Decimal digits

  0 1 2 3 4 5 6 7 8 9

- Underscore character (_)

These characters are used to form the constants, identifiers, and keywords described later in this chapter.

The C compiler treats uppercase and lowercase letters as distinct characters. For example, if a lowercase *a* is specified in an identifier, you cannot substitute an uppercase *A*; you must use the lowercase letter.

White-Space Characters

The space, tab, line-feed, carriage-return, form-feed, vertical-tab, and new-line characters are called ‘‘white-space characters’’ because they serve the same purpose as the spaces between words and lines on a printed page. These characters separate the items you define, such as constants and identifiers, from other items in a program.

The C compiler ignores white-space characters unless you use them as separators or as components of character constants or string literals. Therefore, you can use extra white-space characters to make a program more readable. The compiler also treats comments as white space. (Comments are described in ‘‘Comments.’’)

Punctuation and Special Characters

The punctuation and special characters in the C character set have various uses, from organizing program text to defining the tasks that the compiler or compiled program will carry out. Table 2.1 lists the punctuation and special characters in the C character set.

Table 2.1
Punctuation and Special Characters

Character	Name	Character	Name
,	Comma	!	Exclamation mark
.	Period	|	Vertical bar
;	Semicolon	/	Forward slash
:	Colon	\	Backslash
?	Question mark	~	Tilde
'	Single quotation mark	+	Plus sign
"	Double quotation mark	#	Number sign
(	Left parenthesis	%	Percent sign
)	Right parenthesis	&	Ampersand
[	Left bracket	^	Caret
]	Right bracket	*	Asterisk
{	Left brace	-	Minus sign
}	Right brace	=	Equal sign
<	Left angle bracket	>	Right angle bracket

These characters have special meanings in C. Their uses are described throughout this manual. Any punctuation character from the representable character set that does not appear in Table 2.1 can be used only in string literals, character constants, comments, and file names in **#include** directives.

Escape Sequences

Strings and character constants can contain ‘‘escape sequences.’’ Escape sequences are character combinations representing white-space and nongraphic characters. An escape sequence consists of a backslash (\) followed by a letter or by a combination of digits.

Escape sequences are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of nonprinting characters and characters that normally have special meanings, such as the double-quotation-mark character ("). Table 2.2 lists the C escape sequences.

Table 2.2

Escape Sequences

Escape Sequence	**Name**
\n	New line
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\a	Bell (alert)
\’	Single quotation mark
\″	Double quotation mark
\\	Backslash
\ddd	ASCII character in octal notation
\x*ddd*	ASCII character in hexadecimal notation

If a backslash precedes a character that does not appear in Table 2.2, the backslash is ignored and the character is represented literally. For example, the pattern \c represents the character *c* in a string literal or character constant. However, the use of lowercase letters in escape sequences is reserved by ANSI for future standardization. Therefore, occurrences of undefined escape sequences, though currently innocuous, could pose future portability problems.

The sequence *\ddd* lets you specify any character in the ASCII (American Standard Code for Information Interchange) character set as a three-digit octal character code. Similarly, the sequence *\xddd* lets you specify any ASCII character as a three-digit hexadecimal character code. For example, you can give the ASCII backspace character as the normal C escape sequence (*\b*), or you can code it as *\010* (octal) or *\x008* (hexadecimal).

You can use only the digits 0 through 7 in an octal escape sequence. Though you do not need to use all three digits (as in the form shown in the previous paragraph), you must use at least one. For example, you can specify the ASCII backspace character in octal notation as *\10*. Similarly, you must use at least one digit for a hexadecimal escape sequence, but you can omit the second and third digits. Therefore you could specify the hexadecimal escape sequence for the backspace character either as *\x08* or as *\x8*.

Note

> When you use octal and hexadecimal escape sequences in strings, it is safest to give all three digits of the escape sequence. If you don't specify all digits of the escape sequence, and the character immediately following the escape sequence happens to be an octal or hexadecimal digit, the compiler interprets that character as part of the sequence. For example, if you printed the string *"\x07Bell"*, the result would be *{ell* because *\x07B* is interpreted as the ASCII left-brace character (*{*). The string *\x007Bell* (note the two leading zeros) is the correct way to represent the bell character followed by the word *Bell*. The string *\x7Bell* would generate a compiler diagnostic message because 7BE hexadecimal is too big a number to fit in one byte.

Escape sequences let you send nongraphic control characters to a display device. For example, the escape character *\033* is often used as the first character of a control command for a terminal or printer. Some escape sequences are device specific. For instance, the vertical tab and form feed (*\v* and *\f*) do not affect screen output, but they do perform appropriate operations for a printer.

You should always represent nongraphic characters by escape sequences in C programs, since using the characters directly may generate compiler diagnostic messages.

You can also use the backslash character (\) as a continuation character. When a new-line character immediately follows the backslash, the compiler ignores the backslash and the new line and treats the next line as part of the previous line. This is useful primarily for preprocessor definitions longer than a single line. In the past this feature was also used to create strings longer than one line. However, the string concatenation feature (see ‘‘String Literals’’) is now preferred for creating long string literals.

Operators

‘‘Operators’’ are symbols (both single characters and character combinations) that specify how values are to be manipulated. Each symbol is interpreted as a single unit, called a ‘‘token.’’ (Tokens are defined in ‘‘Tokens.’’)

Table 2.3 lists the symbols that make up the C unary operators and names each operator. Table 2.4 lists the C binary and ternary operators and names them. You must specify operators exactly as they appear in the tables, with no white space between the characters of multicharacter operators. Note that three operator symbols (asterisk, minus sign, and ampersand) appear in both tables. Their interpretation as unary or binary depends on the context in which they appear. The **sizeof** operator is not included in these tables. It consists of a keyword (**sizeof**) rather than a symbol, and is listed in ‘‘Keywords.’’

Table 2.3

Unary Operators

Operator	**Name**
!	Logical NOT
~	Bitwise complement
-	Arithmetic negation
*	Indirection
&	Address of
+	Unary plus[a]

---

[a] The unary plus operator is implemented syntactically, but not semantically.

Table 2.4
Binary and Ternary Operators

Operator	Name	Operator	Name
+	Addition	&&	Logical AND
-	Subtraction		
*	Multiplication	//	Logical OR
/	Division	,	Sequential evaluation
%	Remainder	?:	Conditional[a]
<<	Left shift	++	Increment
>>	Right shift	--	Decrement
<	Less than	=	Simple assignment
<=	Less than or equal to	+=	Addition assignment
>	Greater than	-=	Subtraction assignment
>=	Greater than or equal to	*=	Multiplication assignment
==	Equality	/=	Division assignment
!=	Inequality	%=	Remainder assignment
&	Bitwise AND	>>=	Right-shift assignment
/	Bitwise inclusive OR	<<=	Left-shift assignment
^	Bitwise exclusive OR	&=	Bitwise-AND-assignment
/=	Bitwise inclusive-OR assignment	^=	Bitwise exclusive-OR assignment

[a] The conditional operator is a ternary operator, not a multicharacter operator. A conditional expression has the following form: *expression* **?** *expression* **:** *expression.*

For a complete description of each operator, see the ''Expressions and Assignments'' chapter.

Constants

A *constant* is a number, character, or character string that can be used as a value in a program. A constant's value cannot be modified.

The C language has four kinds of constants: integer constants, floating-point constants, character constants, and string literals.

Integer Constants

Syntax

digits

0*odigits*

0x*hdigits*
0X*hdigits*

An ''integer constant'' is a decimal, octal, or hexadecimal number that represents an integral value in one of the following forms:

- A ''decimal constant'' has the form *digits*, where *digits* represents one or more decimal digits (0 through 9), the first of which is not a zero.
- An ''octal constant'' has the form **0***odigits*, where *odigits* represents one or more octal digits (0 through 7). The leading zero is required.
- A ''hexadecimal constant'' has the form **0x***hdigits* or **0X***hdigits*, where *hdigits* represents one or more hexadecimal digits (0 through 9 and either uppercase or lowercase *a* through *f*. The leading **0x** or **0X** is required.

No white-space characters can separate the digits of an integer constant.

Table 2.5 gives examples of the three forms of integer constants.

Table 2.5

Examples of Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
10	*012*	*0xa* or *0xA*
132	*0204*	*0x84*
32179	*076663*	*0x7db3* or *0x7DB3*

Integer constants always specify positive values. If you need to use a negative value, place a minus sign (-) in front of a constant to form a constant expression with a negative value. (In this case, the minus sign is interpreted as the unary arithmetic negation operator.)

Every integer constant is given a type based on its value. A constant's type determines which conversions must be performed when the constant is used in an expression or when the minus sign (-) is applied, as summarized in the following rules:

- Decimal constants are considered signed quantities and are given **int** type, or **long** type if the size of the value requires it.

- Octal and hexadecimal constants are given **int**, **unsigned int**, **long**, or **unsigned long** type, depending on the size of the constant. If the constant can be represented as an **int**, it is given **int** type. If it is larger than the maximum positive value that can be represented by an **int**, but small enough to be represented in the same number of bits as an **int**, it is given **unsigned int** type. Similarly, a constant that is too large to be represented as an **unsigned int** is given **long** or **unsigned long** type, if necessary.

Table 2.6 shows the ranges of values and the corresponding types for octal and hexadecimal constants on a machine whose **int** type is 16 bits long.

Table 2.6

Types Assigned to Octal and Hexadecimal Constants

Hexadecimal Range	Octal Range	Type
0x0 - 0x7FFF	0 - 077777	**int**
0x8000 - 0xFFFF	0100000 - 0177777	**unsigned int**
0x10000 - 0x7FFFFFFF	0200000 - 017777777777	**long**
0x80000000 - 0xFFFFFFFF	020000000000 - 037777777777	**unsigned long**

The consequence of the typing rules shown in Table 2.6 is that hexadecimal and octal constants are always zero extended when converted to longer types. (For more information on type conversions, see the "Expressions and Assignments." chapter.)

You can force any integer constant to be given **long** type by appending the letter *l* or *L* to the end of the constant. Table 2.7 illustrates some forms of **long** integer constants.

Table 2.7

Examples of Long Integer Constants

Decimal Constants	**Octal Constants**	**Hexadecimal Constants**
10L	*012L*	*0xaL* or *0xAL*
79l	*0115l*	*0x4fl* or *0x4Fl*

Types are described in the "Declarations" chapter and conversions are described in the "Expressions and Assignments" chapter.

Floating-Point Constants

Syntax

[*digits*][*.digits*][**E**|**e**[-|+]*digits*]

A "floating-point constant" is a decimal number that represents a signed real number. The value of a signed real number includes an integer portion, a fractional portion, and an exponent. The *digits* are zero or more decimal digits (0 through 9), and **E** (or **e**) is the exponent symbol. You can omit either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion), but not both. You can leave out the decimal point only if you include an exponent.

The exponent consists of the exponent symbol (**E** or **e**) followed by a constant integer value. The integer value may be negative. No white-space characters can separate the digits or characters of the constant.

Floating-point constants always specify positive values. However, you can place a minus sign (-) in front of the constant to form a constant floating-point expression with a negative value. In this case, the minus sign is treated as an arithmetic operator.

All floating-point constants have type **double**.

Examples

The following examples illustrate some forms of floating-point constants and expressions:

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25E-4
```

You can omit the integer portion of the floating-point constant, as shown in the following examples:

```
-.125
-.175E-2
```

Character Constants

Syntax

'char'

A ''character constant'' is formed by enclosing a single character from the representable character set within single quotation marks (' '). An escape sequence is regarded as a single character and is therefore valid in a character constant. Note that escape characters must be represented by escape sequences or diagnostic messages will be generated. The value of a character constant is the numerical value of the character.

In the syntax above, *char* can be any character from the representable character set (including any escape sequence) except a single quotation mark ('), backslash (\), or new-line character (**\n**). To use a single quotation mark or backslash character as a character constant, precede it with a backslash, as shown in Table 2.8.

Table 2.8

Examples of Character Constants

Constant	Value
’ ’	Single blank space
’a’	Lowercase a
’?’	Question mark
’\b’	Backspace
’\x1B’	ASCII escape character
’\’’	Single quotation mark
’\\’	Backslash

Character constants have type **int**, and are therefore sign extended in type conversions. (See “Type Conversions,” for more information.)

String Literals

Syntax

"*characters*" ["*characters*"]...

A “string literal” is a sequence of characters from the representable character set enclosed in double quotation marks (" "). This example demonstrates a simple string literal:

```
"This is a string literal."
```

In a string literal, *characters* is a placeholder for zero or more characters from the representable character set, including any escape sequence. The double quotation mark ("), backslash (\), or new line must be represented by their escape sequences (\", \\, and **\n**). Non-printing characters should always be represented by a corresponding escape sequence. Each escape sequence is considered a single character.

To force a new line within a string literal, enter the new-line (**\n**) escape sequence at the point in the string where you want the line broken, as follows:

```
"Enter a number between 1 and 100\nOr press Return"
```

The traditional way to form string literals that take up more than one line is to type a backslash, then press RETURN. The backslash causes the compiler to ignore the following new-line character. For example, the string literal

```
"Long strings can be bro\
ken into two or more pieces."
```

is identical to the string

```
"Long strings can be broken into two or more pieces."
```

Two or more string literals separated only by white space will be concatenated into a single string. For example, long strings passed as literals to the **printf** function can now be continued in any column of a succeeding line without affecting their appearance when output, if entered as follows:

```
printf ("This is the first half of the string,"
           " this is the second half") ;
```

As long as each part of the string is enclosed in double quotation marks, the parts will be concatenated and output as a single string:

```
This is the first half of the string, this is the second half
```

You can use string concatenation anywhere you might previously have used a backslash followed by a new-line character to enter strings longer than one line. Because ensuing strings can start in any column of the source code without affecting their on-screen representation, strings can be positioned to enhance source-code readability. For example, the following pointer, initialized as two distinct string literals separated only by white space, is stored as a single string. When properly referenced, as in the following example, it produces a result identical to the previous example:

```
char *string = "This is the first half of the string,"
             " this is the second half" ;

printf("%s" , string) ;
```

To use a double quotation mark or backslash within a string literal, precede it with a backslash as shown in the following examples:

```
"First\\Second"

"\"Yes, I do,\" she said."
```

Note that an escape sequence (such as \\ or \") within a string literal counts as a single character.

The characters of a string are stored in order at contiguous memory locations. A null character (represented by the **\0** escape sequence) is automatically appended to, and marks the end of, each string literal. Each string in a program is generally considered to be distinct; however, two identical strings are not guaranteed to receive separate storage. Therefore, programs should not be designed to allow modification of string literals during execution.

String literals have type array of **char** (**char [ ]**). This means that a string is an array with elements of type **char**. The number of elements in the array is equal to the number of characters in the string, plus one for the terminating null character.

Identifiers

Syntax

letter|_[*letter*|*digit*|_]...

''Identifiers'' are the names you supply for variables, types, functions, and labels in your program. An identifier is a sequence of one or more letters, digits, or underscores (_) that begins with a letter or underscore. Identifiers can contain any number of characters, but only the first 31 characters are significant to the compiler. (Other programs that read the compiler output, such as the linker, may recognize even fewer characters.)

You create an identifier by specifying it in the declaration of a variable, type, or function. You can then use the identifier in later program statements to refer to the associated item. Although statement labels are a special kind of identifier and have their own naming class, their creation is similar to that of variables and functions. (Declarations are described in the ''Declarations'' chapter. Statement labels are described in the ''Statements'' chapter.)

Because the C compiler considers uppercase and lowercase letters distinct characters, you can create distinct identifiers that have the same spelling but different cases for one or more of the letters.

An identifier cannot have the same spelling and case as a keyword of the language. Keywords are described in ''Keywords.''

You should not use leading underscores in identifiers you create: identifiers beginning with an underscore can cause conflicts with the names of system routines or variables, and produce errors. Programs containing names beginning with leading underscores are not guaranteed to be portable.

Note

> Some linkers may further restrict the number and type of characters for globally visible symbols. (Visibility is defined in ''Lifetime and Visibility.'') Also the linker, unlike the compiler, may not distinguish between uppercase and lowercase letters. Consult your linker documentation for information about naming restrictions imposed by the linker.

Examples

The following are examples of identifiers:

```
j
cnt
temp1
top_of_page
skip12
```

Since uppercase and lowercase letters are considered distinct characters, each of the following identifiers is unique:

```
add
ADD
Add
aDD
```

Keywords

"Keywords" are predefined identifiers that have special meanings to the C compiler. They can be used only as defined. The name of a program item cannot have the same spelling and case as a C keyword.

The C language has the following keywords:

auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while

You cannot redefine keywords. However, you can specify text to be substituted for keywords before compilation by using C preprocessor directives (see the "Functions" chapter).

The **volatile** keyword is implemented syntactically, but currently has no semantics associated with it. You cannot use **volatile** as a variable name in your programs.

The following identifiers may be keywords in some implementations. See your compiler guide for more information.

cdecl
far
fortran
huge
near
pascal

Comments

Syntax

/ characters */*

A ''comment'' is a sequence of characters that is treated as a single white-space character by the compiler, but is otherwise ignored. In a comment, *characters* can include any combination of characters from the representable character set, including new-line characters, but excluding the ''end comment'' delimiter (***/**). Comments can occupy more than one line, but they cannot be nested.

Comments can appear anywhere a white-space character is allowed. Since the compiler treats a comment as a single white-space character, you cannot include comments within tokens. However, since the compiler ignores the characters of the comment, you can include keywords in comments without producing errors.

To suppress compilation of a large portion of a program or a program segment that contains comments, bracket the desired portion of code with the **#if** and **#endif** preprocessor directives, rather than ''commenting out'' the code (see ''Conditional Compilation'').

Examples

The following examples illustrate some comments:

```
/* Comments can separate and document
   lines of a program. */

/* Comments can contain keywords such as for
   and while. */

/*****************************************
   Comments can occupy several lines.
 *****************************************/
```

Since comments cannot contain nested comments, the following example causes an error:

```
/* You cannot /* nest */ comments */
```

The error occurs because the compiler recognizes the first */, after the word *nest*, as the end of the comment. It tries to process the remaining text and produces an error when it cannot do so.

Tokens

In a C source program, the basic element recognized by the compiler is the character group known as a ''token.'' A token is source-program text the compiler will not attempt to further analyze into component elements. For example, the following program fragment uses the word *elsewhere* as the name of a function. Although **else** is a keyword in C, there is no confusion between the function name token and the C keyword token it contains.

```
main()
{
      int i = 0;
        if (i)
        elsewhere() ;
}
```

However, if you were to type *elsewhere* as *else where* with a space between *else* and *where*, the preceding example would elicit a compiler diagnostic message noting the lack of a semicolon before the **else** keyword.

The operators, constants, identifiers, and keywords described in this chapter are examples of tokens. Punctuation characters such as brackets ([ ]), braces ({ }), angle brackets (< >), parentheses, and commas are also tokens.

Tokens are delimited by white-space characters and by other tokens, such as operators and punctuation characters. To prevent the compiler from breaking an item down into two or more tokens, white-space characters are not permitted within an identifier, multicharacter operator, or keyword.

When the compiler interprets tokens, it includes as many characters as possible in a single token before moving on to the next token. Because of this behavior, the compiler may not interpret tokens as you intended if they are not properly separated by white space.

Example

Consider the following expression:

```
i+++j
```

In this example, the compiler first makes the longest possible operator (++) from the three plus signs, then processes the remaining plus sign as an addition operator (+). Thus, the expression is interpreted as *(i++) + (j)*, not *(i) + (++j)*. In this and similar cases, use white space and parentheses to avoid ambiguity and ensure proper expression evaluation.

Chapter 3

Program Structure

Introduction 3-1

Source Program 3-2

Source Files 3-4

Functions and Program Execution 3-6

Lifetime and Visibility 3-7
    Blocks 3-7
    Lifetime 3-7
    Visibility 3-8
    Summary 3-9

Naming Classes 3-12

Introduction

This chapter defines terms used later in this manual to describe the C language, and discusses the structure of C source programs. It gives an overview of features of C that are described in detail in other chapters. The syntax and meaning of declarations and definitions are discussed in the ''Declarations'' chapter and the chapter on ''Functions.'' The C preprocessor and pragmas are described in ''Preprocessor Directives and Pragmas.''

Source Program

A C ''source program'' is a collection of any number of directives, pragmas, declarations, definitions, and statements. These constructs are discussed briefly in the following paragraphs. To be valid constructs in Microsoft C, each must have the syntax described in this manual, though they can appear in any order in the program (subject to the rules outlined throughout this manual). However, order of appearance does affect how variables and functions can be used in a program. (See ''Lifetime and Visibility,'' for more information.)

Directives

A ''directive'' instructs the C preprocessor to perform a specific action on the text of the program before compilation.

Pragmas

A ''pragma'' instructs the compiler to perform a particular action at compile time.

Declarations and Definitions

A ''declaration'' establishes an association between the name and the attributes of a variable, function, or type. In C, all variables must be declared before being used.

A ''definition'' of a variable establishes the same associations as a declaration, but also causes storage to be allocated for the variable. Therefore, all definitions are implicitly declarations, but not all declarations are definitions. For example, variable declarations that begin with the **extern** storage-class specifier are ''referencing,'' rather than ''defining,'' declarations. Referencing declarations do not cause storage to be allocated and cannot be initialized. (For more information on storage classes, see the ''Declarations'' chapter.)

''Function declarations'' (or ''prototypes'') establish the name of the function, its return type, and, optionally, its formal parameters. A function definition includes the same elements as the prototype, plus the function body. If you do not supply an explicit declaration for a function, the compiler constructs a prototype from whatever information is available in the first reference to the function, whether that is a definition or a call.

Both function and variable declarations may appear inside or outside a function definition. Any declaration within a function definition is said to appear at the ''internal'' (local) level. A declaration outside all function definitions is said to appear at the ''external'' (global) level.

Variable definitions, like declarations, can appear at the internal level or at the external level. Function definitions always occur at the external level.

Note that declarations of types (for example, structure, union, and **typedef** declarations) that do not include the name of a variable of the type being declared do not cause storage allocation.

Example

The following example illustrates a simple C source program. This source program defines the function named *main* and declares the function named *printf* with a prototype. The program uses defining declarations to initialize the global variables *x* and *y*. The local variables *z* and *w* are declared, but not initialized. Storage is allocated for all these variables, but only *x, y, u,* and *v* contain meaningful values when declared because they are initialized either explicitly or implicitly. The values in *z* and *w* are not meaningful until values are assigned to them in the executable statements.

```
int x = 1;                        /* Defining declarations */
int y = 2;                     /* of external variables */

extern int printf(char *,...);    /* Function "prototype"
                                   or declaration        */

main ()                           /* Function definition
                                     for main function      */
{
        int z;                    /* Definitions for        */
        int w;                 /* two uninitialized       */
                              /* local variables        */

       static int v;          /* Definition of variable
                                 with global lifetime  */
       extern int u;          /* Referencing declaration
                                 of external variable
                                 defined elsewhere      */

        z = y + x;                /* Executable statements */
        w = y - x;
        printf("z= %d  w= %d", z, w);
       printf("v= %d  u= %d", v, u);
}
```

Source Files

A source program can be divided into one or more ‘‘source files.’’ A C source file is a text file containing all or part of a C source program. (For example, a source file may contain just a few of the functions that the program needs.) When you compile a program, you must separately compile, and then link, the individual source files composing the total program. You can also use the **#include** directive to combine separate source files into larger source files before you compile. (For information on ‘‘include’’ files, see the ‘‘Preprocessor Directives and Pragmas’’ chapter.)

A source file can contain any combination of complete directives, pragmas, declarations, and definitions. You cannot split items such as function definitions or large data structures between source files. The last character in a source file must be a new-line character.

A source file need not contain executable statements. For example, you may find it useful to place definitions of variables in one source file and then declare references to these variables in other source files that use them. This technique makes the definitions easy to find and change. For the same reason, manifest constants and macros are often organized into separate include files that may be referenced in source files as required.

Directives in a source file apply only to that source file and its include files. Moreover, each directive applies only to the part of the file that follows the directive. To apply a common set of directives to a whole source program, you must include the directives in all source files that make up the program.

Pragmas usually affect a specific region of a source file. The implementation determines the specific compiler action that a pragma defines. (Your compiler guide describes the effects of particular pragmas.)

Example

The following example illustrates a C source program contained in two source files. Once you have compiled these source files, you can link and then execute them as a single program.

The *main* and *max* functions are assumed to be in separate files, and execution of the program is assumed to begin with the *main* function.

```
/*********************************************************
        Source file 1 - main function
 *********************************************************/

#define ONE     1
#define TWO     2
#define THREE   3

extern int max(int a, int b);    /* Function prototype  */

main ()                          /* Function definition */
{
        int w = ONE, x = TWO, y = THREE;
        int z = 0;
        z = max(x,y);
        w = max(z,w);
}
```

In Source file 1 (above), a prototype of the *max* function is declared. This kind of declaration is sometimes called a ''forward declaration.'' The definition for the *main* function includes calls to *max*.

The lines beginning with a number sign *(#)* are preprocessor directives. These directives tell the preprocessor to replace the identifiers *ONE*, *TWO*, and *THREE* with the corresponding number throughout Source file 1. However, the directives do not apply to Source file 2 (follows), which will be separately compiled and then linked with Source file 1.

```
/*********************************************************
        Source file 2 - definition of max function
 *********************************************************/

int max (int a, int b)        /* Note formal parameters are
                               included in function header */
  {
        if ( a > b )
                return (a);
        else
                return (b);
  }
```

Source file 2 contains the function definition for *max*. This definition satisfies the calls to *max* in Source file 1. Note that the definition for *max* follows the form specified in the ANSI C standard. For more information on this new form and function prototyping, see the ''Functions'' chapter.

Functions and Program Execution

Every C program has a primary function that must be named **main**. The **main** function serves as the starting point for program execution. It usually controls program execution by directing the calls to other functions in the program. A program usually stops executing at the end of **main**, although it can terminate at other points in the program for a variety of reasons depending on the execution environment.

The source program usually has more than one function, with each function designed to perform one or more specific tasks. The **main** function can call these functions to perform their respective tasks. When **main** calls another function, it passes execution control to the function so that execution begins at the first statement in the function. The function returns control when a **return** statement is executed or when the end of the function is reached.

You can declare any function, including **main**, to have parameters. When one function calls another, the called function receives values for its parameters from the calling function. These values are called ''arguments.'' You can declare formal parameters to **main** so that it can receive values from outside the program. (Most commonly, these arguments are passed from the command line when the program is executed.)

Traditionally, the first three parameters of the **main** function are declared to have the names *argc*, *argv*, and *envp*. The *argc* parameter is declared to hold the total number of arguments passed to **main**. The *argv* parameter is declared as an array of pointers, each element of which points to a string representation of an argument passed to **main**. The *envp* parameter is a pointer to a table of string values that set up the environment in which the program executes.

The operating system supplies values for the *argc*, *argv*, and *envp* parameters, and the user supplies the actual arguments to **main**. The operating system, not the C language, determines the argument-passing convention used on a particular system. For more information, see your compiler guide.

If you declare formal parameters to a function, you must declare them when you define the function. Function declarations are described in the ''Declarations,'' and ''Functions'' chapters, including function definitions.

Lifetime and Visibility

To understand how a C program works, you must understand the rules that determine how variables and functions can be used in the program. Three concepts are crucial to understanding these rules: the ''block'' (or compound statement), ''lifetime'' (sometimes called extent), and ''visibility'' (sometimes called scope).

Blocks

A block is a sequence of declarations, definitions, and statements enclosed within braces. There are two types of blocks in C. The ''compound statement'' (discussed more fully in the ''Statements'' chapter) is one type of block. The other, the ''function definition,'' consists of a compound statement comprising the function body plus the function's associated ''header'' (the function name, return type, and formal parameters). A block may encompass other blocks, with the exception that no block can contain a function definition. A block within other blocks is said to be ''nested'' within the encompassing blocks.

Note that, while all compound statements are enclosed within braces, not everything enclosed within braces constitutes a compound statement. For example, though the specifications of array, structure, or enumeration elements may appear within braces, they are not considered compound statements.

Lifetime

''Lifetime'' is the period, during execution of a program, in which a variable or function exists. All functions in a program exist at all times during its execution.

Lifetime of a variable may be internal (local) or external (global). An item with a local lifetime (a ''local item'') has storage and a defined value only within the block where the item is defined or declared. A local item is allocated new storage each time the program enters that block, and it loses its storage (and hence its value) when the program exits the block. If the lifetime of the variable is global (a ''global item''), it has storage and a defined value for the entire duration of a program.

The following rules specify whether a variable has local or global lifetime:

- Variables declared at the internal level (that is, within a block) usually have local lifetimes. You can ensure global lifetime for a variable within a block by including the **static** storage class specifier in its declaration. Once declared **static**, the variable will retain its value from one entry of the block to the next. However, it will still be ''visible'' only within its own block and blocks nested within its own block. (Visibility of objects is discussed in the next section. For information on storage-class specifiers, see the ''Declarations'' chapter.)

- Variables declared at the external level (that is, outside all blocks in the program) always have global lifetimes.

Visibility

An item's ''visibility'' determines the portions of the program in which it can be referenced by name. An item is visible only in portions of a program encompassed by its ''scope,'' which may be limited (in order of increasing restrictiveness) to the file, function, block, or function prototype in which it appears.

In C, only a label name is always confined to function scope. (For more information on labels and label names, see the chapter on ''Statements.'') The scope of any other item is determined by the level at which its declaration occurs. An item declared at the external level has file scope and is visible everywhere within the file. If its declaration occurs within a block (including the list of formal parameters in a function definition), the item's scope is limited to that block and blocks nested within that block. Formal parameter names declared in the parameter list of a function prototype have scope only from the completion of the parameter declaration to the end of the function declarator.

Note

Although an item with a global lifetime *exists* throughout the execution of the source program (for example, an externally declared variable or a local variable declared with the **static** keyword), it may not be visible in all parts of the program.

An item is said to be ''globally visible'' if it is visible, or if you can use appropriate declarations to make it visible, in all the source files making up the program. (For more information on visibility between source files, also known as ''linkage,'' see the chapter on ''Declarations.'')

The following rules govern the visibility of variables and functions within a program:

- Variables declared or defined at the global level (that is, outside all blocks in the program) are visible from their point of definition or declaration to the end of the source file. You can use appropriate declarations to make such variables visible in other source files, as described in ''Storage Classes.'' However, variables declared at the global level with the **static** storage-class specifier are visible only within the source file in which they are defined.

- In general, variables declared or defined at the local level (that is, within a block) are visible only from their point of declaration or definition to the end of the block actually containing the definition or declaration.

- Variables from outer blocks (including those declared at the global level) are visible in all inner blocks. However, the visibility of variables is said to ''nest'' within blocks. For instance, a block within another block can contain declarations for variables whose identifiers (names) are the same as variables in enclosing blocks. Such redefinitions prevail only within the inner block, however. Outer-block definitions are restored as the inner blocks are exited.

- Functions with **static** storage class are visible only in the source file in which they are defined. All other functions are globally visible. (For more information on function declarations, see ''Function Definitions (Prototypes).''

Summary

Table 3.1 summarizes the main factors determining lifetime and visibility of variables and functions. However, the table does not cover all possible cases. For more information, see the ''Declarations'' chapter.

Note

A Microsoft extension to the ANSI C standard provides that functions declared at an internal level may have global visibility. This feature should not be relied upon where portability of source code is a consideration. See your compiler guide for information on enabling Microsoft extensions.

Table 3.1

Summary of Lifetime and Visibility

Level	Item	Storage Class Specifier	Lifetime	Visibility
External	Variable definition	**static**	Global	Restricted to source file in which it occurs
	Variable declaration	**extern**	Global	Remainder of source file
	Function prototype or definition	**static**	Global	Restricted to single source file
	Function prototype	**extern**	Global	Remainder of source file
Internal	Variable declaration	**extern**	Global	Block
	Variable definition	**static**	Global	Block
	Variable definition	**auto** or **register**	Local	Block

Example

The following program example illustrates blocks, nesting, and visibility of variables. In the example, there are four levels of visibility: the external level and three block levels. Assuming that the function *printf* is defined elsewhere in the program, the values will be printed to the screen as noted in the comments preceding each statement.

```
#include <stdio.h>

/* i defined at external level: */
int i = 1;

/* main function defined at external level: */
main ()
{

    /* prints 1 (value of external level i): */
    printf("%d\n", i);

    /* begin first nested block: */
    {

        /* i and j defined at internal level: */
        int i = 2, j = 3;

        /* prints 2, 3: */
        printf("%d\n%d\n", i, j);

        /* begin second nested block: */
        {

            /* i is redefined: */
            int i = 0;

            /* prints 0, 3: */
            printf("%d\n%d\n", i, j);

        /* end of second nested block: */
        }

        /* prints 2 (outer definition restored): */
        printf("%d\n", i);

    /* end of first nested block: */
    }

    /* prints 1 (external level definition restored): */
    printf("%d\n", i);
}
```

Naming Classes

In any C program, identifiers are used to refer to many different kinds of items. When you write a C program, you provide identifiers for the functions, variables, formal parameters, union members, and other items the program uses. C lets you use the same identifier for more than one program item, as long as you follow the rules outlined in this section. (For a definition of an identifier, see the ''Elements of C'' chapter.)

The compiler sets up ''naming classes'' to distinguish between the identifiers used for different kinds of items. The names within each class must be unique to avoid conflict, but an identical name can appear in more than one naming class. This means that you can use the same identifier for two or more different items, provided that the items are in different naming classes. The compiler can resolve references based on the context of the identifier in the program.

The following list describes the kinds of items you can name in C programs and the rules for naming them:

Variables and functions	The names of variables and functions are in a naming class with formal parameters, **typedef** names and enumeration constants. Therefore, variable and function names must be distinct from other names in this class that have the same visibility. However, you can redefine variable and function names within program blocks, as described in ''Lifetime and Visibility.''
Formal parameters	The names of formal parameters to a function are grouped with the names of the function's variables, so the formal parameter names should be distinct from the variable names. You cannot redeclare the formal parameters at the top level of the function. However, the names of the formal parameters may be redefined (that is, used to refer to different items) within subsequent blocks nested within the function body.

Enumeration constants	Enumeration constants are in the same naming class as variable and function names. This means that the names of enumeration constants must be distinct from all variable and function names with the same visibility, and distinct from the names of other enumeration constants with the same visibility. However, like variable names, the names of enumeration constants have nested visibility, so you can redefine them within blocks. (Nested visibility is discussed in ''Lifetime and Visibility.'')
typedef names	The names of types defined with the **typedef** keyword are in a naming class with variable and function names. Therefore, **typedef** names must be distinct from all variable and function names with the same visibility, as well as from the names of formal parameters and enumeration constants. Like variable names, names used for **typedef** types can be redefined within program blocks. See ''Lifetime and Visibility.''
Tags	Enumeration, structure, and union tags are grouped in a single naming class. These tags must be distinct from other tags with the same visibility. Tags do not conflict with any other names.
Members	The members of each structure and union form a naming class. The name of a member must, therefore, be unique within the structure or union, but it does not have to be distinct from other names in the program, including the names of members of different structures and unions.
Statement labels	Statement labels form a separate naming class. Each statement label must be distinct from all other statement labels in the same function. Statement labels do not have to be distinct from other names or from label names in other functions.

Example

Since structure tags, structure members, and variable names are in three different naming classes, the three items named *student* in the following example do not conflict. The context of each item allows correct interpretation of each occurrence of *student* in the program.

For example, when *student* appears after the **struct** keyword, the compiler recognizes it as a structure tag. When *student* appears after a member-selection operator (**->** or **.**), the name refers to the structure member. In other contexts, *student* refers to the structure variable.

```
struct student {
        char student[20];
        int class;
        int id;
        } student;
```

Chapter 4

Declarations

Introduction 4-1

Type Specifiers 4-2
  Storage for Fundamental Types 4-5
  Range of Values 4-7
  Data-Type Categories 4-7

Declarators 4-9
  Array, Pointer, and Function Declarators 4-9
  Complex Declarators 4-10
  Declarators with Special Keywords 4-14

Variable Declarations 4-17
  Simple Variable Declarations 4-18
  Enumeration Declarations 4-19
  Structure Declarations 4-21
  Union Declarations 4-25
  Array Declarations 4-26
  Pointer Declarations 4-28

Function Declarations (Prototypes) 4-33
  Formal Parameters 4-34
  Return Type 4-34
  The List of Formal Parameters 4-35
  Summary 4-36

Storage Classes 4-40
  Variable Declarations at the Global Level 4-41
  Variable Declarations at the Local Level 4-44
  Function Declarations at the Global and Local Levels 4-46

Initialization 4-48
  Fundamental and Pointer Types 4-49
  Aggregate Types 4-50
  String Initializers 4-54

Type Declarations 4-55
    Structure, Union, and Enumeration Types 4-55
    Using typedef Declarations 4-56

Type Names 4-58

Introduction

This chapter describes the form and constituents of C declarations for variables, functions, and types. C declarations have the form

[*sc-specifier*] [*type-specifier*] *declarator*[=*initializer*] [,*declarator*[=*initializer*]]...

where *sc-specifier* is a storage-class specifier; *type-specifier* is the name of a defined type; and *initializer* gives the value or sequence of values to be assigned to the variable being declared. The *declarator* is an identifier that can be modified with brackets ([ ]), asterisks (*), or parentheses (( )).

You must explicitly declare all C variables before using them. You can declare a C function explicitly with a function prototype. If you do not provide a prototype, one is created automatically from whatever information is included in the first reference to the function, whether that reference is a definition or a call.

The C language includes a standard set of data types. You can add your own data types by declaring new ones based on types already defined. You can declare arrays, data structures, and pointers to both variables and functions.

C declarations require one or more ''declarators.'' A declarator is an identifier that can be modified with brackets ([ ]), asterisks (*), or parentheses (( )) to declare an array, pointer, or function type, respectively. When you declare simple variables (such as character, integer, and floating-point items), or structures and unions of simple variables, the declarator is just an identifier.

Four storage-class specifiers are defined in C: **auto**, **extern**, **register**, and **static**. The storage-class specifier of a declaration affects how the declared item is stored and initialized and which parts of a program can reference the item. Location of the declaration within the source program and the presence or absence of other declarations of the variable are also important factors in determining the visibility of variables.

Function prototype declarations are presented in ''Function Declarations (Prototypes)'' in this chapter and in the ''Functions'' chapter of this guide. For information on function definitions, see the ''Functions'' chapter.

Type Specifiers

The C language provides definitions for a set of basic data types, called ''fundamental'' types. Their names are listed in Table 4.1.

Table 4.1

Fundamental Types

Integral Types[a]	Floating-Point Types	Other
char	**float**	**void**[c]
int	**double**	**const**
short	**long double**[b]	**volatile**[d]
long		
signed		
unsigned		
enum		

[a] The optional keywords **signed** and **unsigned** can precede any of the integral types, except **enum**, and can also be used alone as type specifiers, in which case they are understood as **signed int** and **unsigned int**, respectively. When used alone, the keyword **int** is assumed to be **signed**. When used alone, the keywords **long** and **short** are understood as **long int** and **short int**.

[b] The **long double** type is semantically equivalent to **double**, but is syntactically distinct.

[c] The keyword **void** has three uses: as a function return type, as an argument-type list for a function that will take no arguments, and to modify a pointer.

[d] The **volatile** keyword is implemented syntactically, but not semantically.

Enumeration types are considered fundamental types.

Note

> The **long float** type is no longer supported, and occurrences of it in old code should be changed to **double**.

The **signed char**, **signed int**, **signed short int**, and **signed long int** types, together with their **unsigned** counterparts and **enum**, are called ''integral'' types. The **float**, **double**, and **long double** type specifiers are referred to as ''floating'' or ''floating-point'' types. You can use any integral or floating-point type specifier in a variable or function declaration.

You can use the **void** type to declare functions that return no value or to declare a pointer to an unspecified type. When the keyword **void** occurs alone within the parentheses following a function name, it is not interpreted as a type specifier. In that context **void** indicates only that the function accepts no arguments. Function types are discussed in ''Function Declarations (Prototypes).''

The **const** type specifier declares an object as nonmodifiable. The **const** keyword can be a modifier for any fundamental or aggregate type, or to modify a pointer to an object of any type. A **const** type specifier can modify a **typedef**. A declaration that includes the keyword **const** as a modifier of an aggregate type declarator indicates that each element of the aggregate type is unmodifiable. If an item is declared with only the **const** type specifier, its type is taken to be **const int**. A **const** object may be placed in a read-only region of storage.

The **volatile** type specifier declares an item whose value may legitimately be changed by something beyond the control of the program in which it appears. The **volatile** keyword can be used in the same circumstances as **const** (previously described). An item can be both **const** and **volatile**, in which case the item could not be legitimately modified by its own program, but could be modified by some asynchronous process. The **volatile** keyword is implemented syntactically, but not semantically.

You can create additional type specifiers with **typedef** declarations (see ''Type Declarations''). When used in a declaration, such specifiers may only be modified by the **const** and **volatile** modifiers.

Type specifiers are commonly abbreviated, as shown in Table 4.2. Integral types are signed by default. Thus, if you omit the **unsigned** keyword from the type specifier, the integral type is signed, even if you do not specify the **signed** keyword.

Type Specifiers

In some implementations, you can specify a compiler option that changes the default **char** type from signed to unsigned. When this option is in effect, the abbreviation **char** means the same as **unsigned char**, and you must use the **signed** keyword to declare a signed character value. Compiler options are described in your compiler guide.

Note

> This manual generally uses the abbreviated forms of the type specifiers listed in Table 4.2 rather than the long forms, and it assumes that the **char** type is signed by default. Therefore, throughout this manual, **char** stands for **signed char**.

Table 4.2

Type Specifiers and Abbreviations

Type Specifier	Abbreviations
signed char[a]	**char**
signed int	**signed, int**
signed short int	**short, signed short**
signed long int	**long, signed long**
unsigned char[b]	--
unsigned int	**unsigned**
unsigned short int	**unsigned short**
unsigned long int	**unsigned long**
float	--
const int	**const**
volatile int	**volatile**
const volatile int	**const volatile**

[a] When you make the **char** type unsigned by default (by specifying the appropriate compiler option), you cannot abbreviate **signed char**.

[b] When you make the **char** type unsigned by default (by specifying the appropriate compiler option), you can abbreviate **unsigned char** as **char**.

Storage for Fundamental Types

Table 4.3 summarizes the storage associated with each fundamental type and gives the range of values that can be stored in a variable of each type. Since the **void** type specifier is only used to denote a function with no return value or a pointer to an unspecified type, it is not included in the table. Similarly, the table does not include **const** or **volatile** because a variable type modified by **const** or **volatile** retains its storage size and can contain any value within range for its fundamental type.

Table 4.3

Storage and Range of Values for Fundamental Types

Type	Storage	Range of Values (Internal)
char	1 byte	–128 to 127
int	implementation defined	
short	2 bytes	–32,768 to 32,767
long	4 bytes	–2,147,483,648 to 2,147,483,647
unsigned char	1 byte	0 to 255
unsigned	implementation defined	
unsigned short	2 bytes	0 to 65,535
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	IEEE-standard notation; discussed below
double	8 bytes	IEEE-standard notation; discussed below
long double	8 bytes	IEEE-standard notation; discussed below

The **char** type stores the integer value of a member of the representable character set. That integer value is the ASCII code corresponding to the specified character. Since the **char** type is interpreted as a signed, 1-byte integer, a **char** variable can store values in the range -128 to 127, although only the values from 0 to 127 have character equivalents. Similarly, an **unsigned char** variable can store values in the range 0-255.

Note that the C language does not define the storage and range associated with the **int** and **unsigned int** types. Instead, the size of a signed or unsigned **int** item is the standard size of an integer on a particular machine. For example, on a 16-bit machine the **int** type is usually 16 bits, or 2 bytes. On a 32-bit machine the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent to either the **short int** or the **long int** type, and the **unsigned int** type is equivalent to either the **unsigned short** or the **unsigned long** type, depending on the implementation.

The type specifiers **int** and **unsigned int** (or simply **unsigned**) define certain features of the C language (for instance, the **enum** type discussed later in ‘‘Type Declarations’’). In these cases, the definitions of **int** and **unsigned int** for a particular implementation determine the actual storage.

Note

> The **int** and **unsigned int** type specifiers are widely used in C programs because they let a particular machine handle integer values in the most efficient way for that machine. However, since the sizes of the **int** and **unsigned int** types vary, programs that depend on a specific **int** size may not be portable to other machines. To make programs more portable, you can use expressions with the **sizeof** operator instead of hard-coded data sizes. The actual sizes of **int** and **unsigned int** are discussed in your compiler guide.

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers, Inc.) format. Values with **float** type have 4 bytes, consisting of a sign bit, an 8-bit excess-127 binary exponent, and a 23-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives a range of approximately 3.4E-38 to 3.4E+38 for type **float**.

Values with **double** type have 8 bytes. The format is similar to the **float** format except that it has an 11-bit excess-1023 exponent and a 52-bit mantissa, plus the implied high-order 1 bit. This format gives a range of approximately 1.7E-308 to 1.7E+308 for type **double**.

Range of Values

The range of values for a variable is bounded by the minimum and maximum values that can be represented *internally* in a given number of bits. However, because of C's conversion rules (discussed in detail in the ''Expressions and Assignments'' chapter), you cannot always use the maximum or minimum value for a constant of a particular type in an expression.

For example, the constant expression *-32768* consists of the arithmetic negation operator (-) applied to the constant value 32,768. Since 32,768 is too large to represent as a **short int**, it is given the **long** type. Consequently, the constant expression *-32768* has **long** type. You can only represent -32,768 as a **short int** by type-casting it to the **short** type. No information is lost in the type cast, since -32,768 can be represented internally in 2 bytes.

Similarly, a value such as 65,000 can only be represented as an **unsigned short** by type-casting the value to **unsigned short** type or by giving the value in octal or hexadecimal notation. The value 65,000 in decimal notation is considered a signed constant. It is given the **long** type because 65,000 does not fit into a **short**. You can cast this **long** value to the **unsigned short** type without loss of information, since 65,000 can fit in 2 bytes when it is stored as an unsigned number.

Octal and hexadecimal constants may have either **signed** or **unsigned** type, depending on their size (see ''Integer Constants,'' for more information). However, the method used to assign types to octal and hexadecimal constants ensures that they always behave like unsigned integers in type conversions.

Data-Type Categories

The C data types fall into two general categories, called scalar and aggregate. Scalar types include pointers and arithmetic types. Arithmetic types include all floating and integral types, as described in this section. Aggregate types include arrays and structures. Table 4.4 illustrates the categories of C data types.

Table 4.4

C Data-Type Categories

Data Types	Categories		
char			
int			
short			
long	*Integral*		
signed	*Types*		
unsigned		*Arithmetic*	
enum		*Types*	*Scalar*
			Types
float			
double	*Floating*		
long double	*Types*		
Pointers			
Arrays			*Aggregate*
Structures			*Types*

Declarators

Syntax

identifier
declarator[[*constant-expression*]]
**declarator*
(*declarator*)

The C language lets you declare ''arrays'' of values, ''pointers'' to values, and ''functions returning'' values of specified types. You must use a ''declarator'' to declare these items.

A ''declarator'' is an identifier that may be modified by brackets ([ ]), asterisks (*), or parentheses (( )) to declare an array, pointer, or function type, respectively. Declarators appear in the array, pointer, and function declarations described later in this chapter. The following section discusses the rules for forming and interpreting declarators.

Array, Pointer, and Function Declarators

When a declarator consists of an unmodified identifier, the item being declared has a base type. If the identifier is followed by brackets ([ ]), the type is modified to an *array* type. If asterisks (*) appear to the left of an identifier, the type is modified to a *pointer* type. If the identifier is followed by parentheses, the type is modified to a *function returning* type.

A declarator must include a type specifier to be a complete declaration. The type specifier gives the type of the elements of an array type, the type of object addressed by a pointer type, or the return type of a function.

The sections on array, pointer, and function declarations later in this chapter discuss each type of declaration in detail.

The following examples illustrate the simplest forms of declarators:

Example 1

This example declares an array of **int** values named *list*:

```
int list[20];
```

Example 2

The following example declares a pointer named *cp* to a **char** value:

```
char *cp;
```

Example 3

The following declares a function named *func*, with no arguments, that returns a **double** value:

```
double func(void);
```

Complex Declarators

You can enclose any declarator in parentheses to specify a particular interpretation of a complex declarator.

A ‘‘complex’’ declarator is an identifier qualified by more than one array, pointer, or function modifier. You can apply various combinations of array, pointer, and function modifiers to a single identifier. However, a declarator may not have the following illegal combinations:

- An array cannot have functions as its elements.
- A function cannot return an array or a function.

In interpreting complex declarators, brackets and parentheses (that is, modifiers to the right of the identifier) take precedence over asterisks (that is, modifiers to the left of the identifier). Brackets and parentheses have the same precedence and associate from left to right. After the declarator has been fully interpreted, the type specifier is applied as the last step. By using parentheses you can override the default association order and force a particular interpretation.

A simple way to interpret complex declarators is to read them from the inside out, using the following four steps:

1. Start with the identifier and look to the right for brackets or parentheses (if any).
2. Interpret these brackets or parentheses, then look to the left for asterisks.

3. For each right parenthesis you encounter, apply rules 1 and 2 to everything within the parentheses.

4. Apply the type specifier.

Example 1

In the following example, the steps are labeled in order and can be interpreted as follows:

1. The identifier *var* is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. **char** values.

```
char *(*(*var)())[10];
 ^   ^ ^ ^ ^  ^  ^
 7   6 4 2 1  3  5
```

Examples 2 through 9 illustrate complex declarations further and show how parentheses can affect the meaning of a declaration.

Example 2

In the following example, the array modifier has higher priority than the pointer modifier, so *var* is declared to be an array. The pointer modifier applies to the type of the array elements; therefore, the array elements are pointers to **int** values.

```
     /* array of pointers to int values */

int *var[5];
```

Example 3

In the following example, parentheses give the pointer modifier higher priority than the array modifier, and *var* is declared to be a pointer to an array of five **int** values.

```
             /* pointer to array of int values */

      int (*var)[5];
```

Example 4

Function modifiers also have higher priority than pointer modifiers, so this example declares *var* to be a function returning a pointer to a **long** value. The function is declared to take two **long** values as arguments.

```
             /* function returning pointer to a long */

      long *var(long,long);
```

Example 5

This example is similar to Example 3. Parentheses give the pointer modifier higher priority than the function modifier, and *var* is declared to be a pointer to a function that returns a **long** value. Again, the function takes two **long** arguments.

```
             /* pointer to function returning long */

      long (*var)(long,long);
```

Example 6

The elements of an array cannot be functions, but this example demonstrates how to declare an array of pointers to functions instead. In this example, *var* is declared to be an array of five pointers to functions that return structures with two members. The arguments to the functions are declared to be two structures with the same structure type, *both.* Note that the parentheses surrounding **var[5]* are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown here:

```
                 /* ILLEGAL */
struct both *var[5]( struct both, struct both );


      /* array of pointers to functions
         returning structures */

struct both {
     int a;
     char b;
     } ( *var[5] )( struct both, struct both );
```

Example 7

This example shows how to declare a function returning a pointer to an array, since functions returning arrays are illegal. Here *var* is declared to be a function returning a pointer to an array of three **double** values. The function *var* takes one argument. The argument, like the return value, is a pointer to an array of three **double** values. The argument type is given by a complex abstract declarator. The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of three pointers to **double** values. For a discussion and examples of abstract declarators, see ‘‘Type Names.’’

```
          /* function returning pointer
             to an array of 3 double values */

     double ( *var( double (*)[3] ) )[3];
```

Example 8

As this example shows, a pointer can point to another pointer, and an array can contain arrays as elements. Here *var* is an array of five elements. Each element is a five-element array of pointers that point to unions, each of which have two members.

```
          /* array of arrays of pointers
             to pointers to unions */

     union sign {
          int x;
          unsigned y;
          } **var[5][5];
```

Example 9

This example shows how the placement of parentheses changes the meaning of the declaration. In this example, *var* is a five-element array of pointers to five-element arrays of pointers to unions.

```
          /* array of pointers to arrays
             of pointers to unions */

     union sign *(*var[5])[5];
```

Declarators with Special Keywords

Your implementation of Microsoft C may include the following special keywords:

1. **cdecl**
2. **far**
3. **fortran**
4. **huge**
5. **near**
6. **pascal**

These keywords modify the meaning of variable and function declarations. See your compiler guide for a full discussion of the effects of these special keywords.

When a special keyword appears in a declarator, it modifies the item immediately to the right of the keyword. You can apply more than one special keyword to the same item. For example, you might modify a function identifier with both the **far** keyword and the **pascal** keyword. In this case, the order of the keywords does not matter (that is, **far pascal** and **pascal far** have the same effect). Thus the ''binding'' characteristics of the special keywords are the same as those of the type specifiers **const** and **volatile**. (The section ''Type Specifiers,'' contains descriptions of the **const** and **volatile** keywords.)

You can also use two or more special keywords in different parts of a declaration to modify the meaning of the declaration. For example, the following declaration contains two occurrences of the **far** keyword:

```
int far * pascal far func(void);
```

In this example, the **pascal** and **far** keywords modify the function identifier *func*. The return value of *func* is declared to be a **far** pointer to an **int** value.

As in any C declaration, you can use parentheses to override the default interpretation of the declaration. The rules governing complex declarators also apply to declarations that use the special keywords.

The following examples show the use of special keywords in declarations.

Example 1

This example declares a **huge** array named *database* with 65,000 **int** elements. The **huge** keyword modifies the array declarator.

```
int huge database[65000];
```

Example 2

In this example, the **far** keyword modifies the asterisk to its right, making *x* a **far** pointer to a pointer to **char**.

```
char * far * x;
```

This declaration is equivalent to the following declaration:

```
char * (far *x);
```

Example 3

This example shows two equivalent declarations. Both declare *calc* as a function with the **near** and **cdecl** attributes.

```
double near cdecl calc(double,double);

double cdecl near calc(double,double);
```

Example 4

Example 4 also shows two declarations. The first declares a **far fortran** array of characters named *initlist*, and the second declares three **far** pointers named *nextchar*, *prevchar*, and *currentchar*. These pointers might be used to store the addresses of characters in the *initlist* array. Note that the **far** keyword must be repeated before each declarator.

```
char far fortran initlist[INITSIZE];

char far *nextchar, far *prevchar, far *currentchar;
```

Example 5

Example 5 shows a more complex declaration with several occurrences of the **far** keyword.

```
char far *(far *getint)(int far *);
 ^   ^      ^      ^   ^     ^
 6   5      2      1   3     4
```

The following procedure would be used to interpret this declaration:

1. The identifier *getint* is declared as a
2. **far** pointer to
3. a function taking
4. a single argument that is a **far** pointer to an **int** value
5. and returning a **far** pointer to a
6. **char** value.

Note that the **far** keyword always modifies the item immediately to its right.

Variable Declarations

Syntax

[*sc-specifier*] *type-specifier declarator* [**,** *declarator*]...

This section describes the form and meaning of variable declarations. In particular, it explains how to declare the following:

Simple variables	Single-value variables with integral or floating-point type
Enumeration variables	Simple variables with integral type that hold one value from a set of named integer constants
Structures	Variables composed of a collection of values that may have different types
Unions	Variables composed of several values of different types, which occupy the same storage space
Arrays	Variables composed of a collection of elements with the same type
Pointers	Variables that point to other variables and contain variable locations (in the form of addresses) instead of values

In the general form of a variable declaration, *type-specifier* gives the data type of the variable and *declarator* gives the name of the variable, possibly modified to declare an array or a pointer type. The *type-specifier* can be a compound, as when the type is modified by **const**, **volatile**, or one of the special keywords described in ''Declarators with Special Keywords.'' You can define more than one variable in a declaration by using multiple declarators, separated by commas. For example, *int const far *fp* declares a variable named *fp* as a far pointer to a nonmodifiable **int** value.

The *sc-specifier* gives the storage class of the variable. In some contexts, you can initialize variables at the time you declare them. For information about storage classes and initialization, see the sections on ''Storage Classes'' and ''Initialization,'' respectively.

Simple Variable Declarations

Syntax

[*sc-specifier*] *type-specifier identifier* [, *identifier*]...;

The declaration of a simple variable specifies the variable's name and type. It can also specify the variable's storage class, as described in "Storage Classes." The *identifier* in the declaration is the variable's name. The *type-specifier* is the name of a defined data type.

You can use a list of identifiers separated by commas (,) to specify several variables in the same declaration. Each identifier in the list names a variable. All variables defined in the declaration have the same type.

Example 1

Example 1 declares a simple variable named *x*. This variable can hold any value in the set defined by the **int** type for a particular implementation. The simple object *y* is declared as a constant value of type **int**. It is initialized to the value 1, and is not modifiable. If the declaration of *y* was for an uninitialized external, it would receive an initial value of 0, and that value would be unmodifiable.

```
int x;
int const y=1;
```

Example 2

This example declares two variables named *reply* and *flag*. Both variables have **unsigned long** type and hold unsigned integral values.

```
unsigned long reply, flag;
```

Example 3

The following example declares a variable named *order* that has **double** type and can hold floating-point values.

```
double order;
```

Enumeration Declarations

Syntax

enum [*tag*] {*enum-list*} [*declarator* [, *declarator*]...];

enum *tag* [*identifier* [, *declarator*]...];

An ''enumeration declaration'' gives the name of an enumeration variable and defines a set of named integer constants (the ''enumeration set''). A variable with enumeration type stores one of the values of the enumeration set defined by that type. The integer constants of the enumeration set have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value.

Variables of **enum** type are treated as if they are of type **int** in all cases. They may be used in indexing expressions and as operands of all arithmetic and relational operators.

Enumeration declarations begin with the **enum** keyword and have the two forms shown at the beginning of this section and described below:

- In the first form, *enum-list* specifies the values and names of the enumeration set. (The *enum-list* is described in detail below.) The optional *tag* is an identifier that names the enumeration type defined by *enum-list*. The *declarator* names the enumeration variable. You can specify zero or more enumeration variables in a single enumeration declaration.

- The second form of the enumeration declaration uses a previously defined enumeration *tag* to refer to an enumeration type defined elsewhere. The *tag* must refer to a defined enumeration type, and that enumeration type must be currently visible. Since the enumeration type is defined elsewhere, *enum-list* does not appear in this type of declaration. Declarations of pointers to enumerations and **typedef** declarations for enumeration types can use the enumeration *tag* before the enumeration type is defined. However, the enumeration definition must be encountered prior to any actual use of the **typedef** declaration or pointer.

If a *tag* argument appears, but no *declarator* is given, the declaration constitutes a declaration for an enumeration tag.

An *enum-list* has the following form:

> *identifier* [= *constant-expression*]
> [, *identifier* [= *constant-expression*] ...]

Each *identifier* in an enumeration list names a value of the enumeration set. By default, the first identifier is associated with the value 0, the next identifier is associated with the value 1, and so on through the last identifier in the declaration. The name of an enumeration constant is equivalent to its value.

The optional phrase = *constant-expression* overrides the default sequence of values. Thus, if *identifier* = *constant-expression* appears in *enum-list*, the identifier is associated with the value given by *constant-expression*. The *constant-expression* must have **int** type and can be negative. The next identifier in the list is associated with the value of *constant-expression* + 1, unless you explicitly associate it with another value.

The following rules apply to the members of an enumeration set:

- An enumeration set can contain duplicate constant values. For example, you could associate the value 0 with two different identifiers named *null* and *zero* in the same set.
- The identifiers in the enumeration list must be distinct from other identifiers with the same visibility, including ordinary variable names and identifiers in other enumeration lists.
- Enumeration tags must be distinct from other enumeration, structure, and union tags with the same visibility.
- A comma is allowed following the last item in the enumeration list.

Example 1

This example defines an enumeration type named *day* and declares a variable named *workday* with that enumeration type. The value 0 is associated with *saturday* by default. The identifier *sunday* is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

```
enum day {
        saturday,
        sunday = 0,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday
        } workday;
```

Example 2

In this example, a value from the set defined in Example 1 is assigned to the variable *today*. Note that the name of the enumeration constant is used to assign the value. Since the *day* enumeration type was previously declared, only the enumeration tag is necessary.

```
enum day today = wednesday;
```

Structure Declarations

Syntax

struct [*tag*] {*member-declaration-list*} [*declarator* [, *declarator*]...];

struct *tag*[*declarator* [, *declarator*]...];

A ‘‘structure declaration’’ names a structure variable and specifies a sequence of variable values (called ‘‘members’’ of the structure) that can have different types. A variable of that structure type holds the entire sequence defined by that type.

Structure declarations begin with the **struct** keyword and have two forms:

- In the first form, a *member-declaration-list* specifies the types and names of the structure members. The optional *tag* is an identifier that names the structure type defined by *member-declaration-list*.

- The second form uses a previously defined structure *tag* to refer to a structure type defined elsewhere. Thus, *member-declaration-list* is not needed as long as the definition is visible. Declarations of pointers to structures and **typedefs** for structure types can use the structure tag before the structure type is defined. However, the structure definition must be encountered prior to any actual use of the **typedef** or pointer.

In both forms, each *declarator* specifies a structure variable. A *declarator* can also modify the type of the variable to a pointer to the structure type, an array of structures, or a function returning a structure. If *tag* is given, but *declarator* does not appear, the declaration constitutes a type declaration for a structure tag.

Structure tags must be distinct from other structure, union, and enumeration tags with the same visibility.

A *member-declaration-list* argument contains one or more variable or bitfield declarations.

Each variable declared in the member-declaration list is defined as a member of the structure type. Variable declarations within the member-declaration list have the same form as other variable declarations discussed in this chapter, except that the declarations cannot contain storage-class specifiers or initializers. The structure members can have any variable type: fundamental, array, pointer, union, or structure.

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears as long as the structure type has a tag. This lets you create linked lists of structures.

A bitfield declaration has the following form:

type-specifier [*identifier*] **:** *constant-expression*;

The *constant-expression* specifies the number of bits in the bitfield. The *type-specifier* has type **int** (**signed** or **unsigned**) and *constant-expression* must be a non-negative integer value. Arrays of bitfields, pointers to bitfields, and functions returning bitfields are not allowed. The optional *identifier* names the bitfield. Unnamed bitfields can be used as ''dummy'' fields, for alignment purposes. An unnamed bitfield whose width is specified as 0 guarantees that storage for the member following it in the member-declaration list begins on an **int** boundary.

Each *identifier* in a member-declaration list must be unique within the list. However, they do not have to be distinct from ordinary variable names or from identifiers in other member-declaration lists.

Note

A Microsoft extension to the ANSI C standard allows **char** and **long** types (both **signed** and **unsigned**) for bitfields. Unnamed bitfields with base type **long** or **char** (**signed** or **unsigned**) force alignment to a boundary appropriate to the base type.

Microsoft C does not implement **signed** bitfields. The syntax is allowed, but a bitfield specified as **signed** is treated as **unsigned** in all conversions.

Storage

Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest. Storage for each member begins on a memory boundary appropriate to its type. Therefore, unnamed spaces (‘‘holes’’) may appear between structure members in memory.

Bitfields are not stored across boundaries of their declared type. For example, a bitfield declared with **unsigned int** type is packed into the space remaining (if any) if the previous bitfield was of type **unsigned int**. Otherwise, it begins a new object on an **int** boundary.

Example 1

This example defines a structure variable named *complex*. This structure has two members with **float** type, *x* and *y*. The structure type has no tag and is therefore unnamed.

```
struct {
           float x,y;
    } complex;
```

Example 2

This example defines a structure variable named *temp*. The structure has three members: *name*, *id*, and *class*. The *name* member is a 20-element array, and *id* and *class* are simple members with **int** and **long** type, respectively. The identifier *employee* is the structure tag.

```
struct employee {
            char name[20];
            int id;
            long class;
    } temp;
```

Example 3

This example defines three structure variables: *student*, *faculty*, and *staff*. Each structure has the same list of three members. The members are declared to have the structure type *employee*, defined in Example 2.

```
struct employee student, faculty, staff;
```

Example 4

This example defines a structure variable named *x*. The first two members of the structure are a **char** variable and a pointer to a **float** value. The third member, *next*, is declared as a pointer to the structure type being defined (*sample*).

```
struct sample {
            char c;
            float *pf;
            struct sample *next;
    } x;
```

Example 5

This example defines a two-dimensional array of structures named *screen*. The array contains 2000 elements. Each element is an individual structure containing four bitfield members: *icon*, *color*, *underline*, and *blink*.

```
struct {
            unsigned icon : 8;
            unsigned color : 4;
            unsigned underline : 1;
            unsigned blink : 1;
    } screen[25][80];
```

Union Declarations

Syntax

union [*tag*] {*member-declaration-list*} [*declarator* [, *declarator*.]..];

union *tag*[*declarator*[, *declarator*]...];

A ''union declaration'' names a union variable and specifies a set of variable values, called ''members'' of the union, that can have different types. A variable with **union** type stores one of the values defined by that type.

Union declarations have the same form as structure declarations, except that they begin with the **union** keyword instead of the **struct** keyword. The same rules govern structure and union declarations, except that bitfield members are not allowed in unions.

Storage

The storage associated with a union variable is the storage required for the largest member of the union. When a smaller member is stored, the union variable may contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member.

Example 1

This example defines a union variable with *sign* type and declares a variable named *number* that has two members: *svar*, a signed integer, and *uvar*, an unsigned integer. This declaration allows the current value of *number* to be stored as either a signed or an unsigned value. The tag associated with this union type is *sign*.

```
union sign {
        int svar;
        unsigned uvar;
    } number;
```

Example 2

This example defines a union variable named *jack*. The members of the union are, in order of their declaration, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage allocated for *jack* is the storage required for the 20-element array *f*, since *f* is the longest member of the union. Because there is no tag associated with the union, its type is unnamed.

```
union {
        char *a, b;
        float f[20];
    } jack;
```

Example 3

This example defines a two-dimensional array of unions named *screen*. The array contains 2000 elements. Each element of the array is an individual union with two members: *window1* and *screenval*. The *window1* member is a structure with two bitfield members, *icon* and *color*. The *screenval* member is an **int**. At any given time, each union element holds either the **int** represented by *screenval* or the structure represented by *window1*.

```
union {
        struct {
                unsigned int icon : 8;
                unsigned color : 4;
        } window1;
        int screenval;
    } screen[25][80];
```

Array Declarations

Syntax

type-specifier declarator [*constant-expression*];
type-specifier declarator [ ];

An "array declaration" names the array and specifies the type of its elements. It may also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements, as described in the section on "Identifiers."

Array declarations have the two forms shown at the beginning of this section. Their syntax differs as follows:

- In the first form, the *constant-expression* argument within the brackets specifies the number of elements in the array. Each element has the type given by *type-specifier*, which can be any type except **void**. An array element cannot be a function type.

- The second form omits the *constant-expression* argument in brackets. You can use this form only if you have initialized the array, declared it as a formal parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

In both forms, *declarator* names the variable and may modify the variable's type. The brackets ([ ]) following *declarator* modify the declarator to array type. You can declare an array of arrays (a ''multidimensional'' array) by following the array declarator with a list of bracketed constant expressions, as shown:

type-specifier declarator[*constant-expression*] [*constant-expression*] ...

Each *constant-expression* in brackets defines the number of elements in a given dimension: two-dimensional arrays have two bracketed expressions, three-dimensional arrays have three, and so on. When you declare a multidimensional array within a function, you can omit the first constant expression if you have initialized the array, declared it as a formal parameter, or declared it as a reference to an array explicitly defined elsewhere in the program.

You can define arrays of pointers to various types of objects by using complex declarators, as described in ''Complex Declarators.''

Storage

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing memory locations, from the first element to the last. No blanks separate the array elements in storage.

Arrays are stored by row. For example, the following array consists of two rows with three columns each:

```
char A[2][3];
```

The three columns of the first row are stored first, followed by the three columns of the second row. This means that the last subscript varies most quickly.

To refer to an individual element of an array, use a subscript expression, as described in ''Subscript Expressions.''

Example 1

This example declares an array variable named *scores* with 10 elements, each of which has **int** type. The variable named *game* is declared as a simple variable with **int** type.

```
int scores[10], game;
```

Example 2

This example declares a two-dimensional array named *matrix*. The array has 150 elements, each having **float** type.

```
float matrix[10][15];
```

Example 3

This example declares an array of structures. This array has 100 elements; each element is a structure containing two members.

```
struct {
            float x,y;
            } complex[100];
```

Example 4

This example declares the type and name of an array of pointers to **char**. The actual definition of *name* occurs elsewhere.

```
extern char *name[];
```

Pointer Declarations

Syntax

type-specifier ***** [*modification-spec*] *declarator*;
extern char *name [ ];

A ‘‘pointer declaration’’ names a pointer variable and specifies the type of the object to which the variable points. A variable declared as a pointer holds a memory address.

The *type-specifier* gives the type of the object, which can be any fundamental, structure, or union type. Pointer variables can also point to functions, arrays, and other pointers. (For information on declaring more complex pointer types, refer to the section on ''Complex Declarators.'')

By making *type-specifier* **void**, you can delay specification of the type to which the pointer refers. Such an item is referred to as a ''pointer to **void**'' (**void ***). A variable declared as a pointer to **void** can be used to point to an object of any type. However, in order to perform operations on the pointer or on the object to which it points, the type to which it points must be explicitly specified for each operation. Such conversion can be accomplished with a type cast.

The *modification-spec* can be either **const** or **volatile**, or both. These specify, respectively, that the pointer will not be modified by the program itself (**const**), or that the pointer may legitimately be modified by some process beyond the control of the program (**volatile**). (For more information on **const** and **volatile**, see ''Type Specifiers.'')

The *declarator* names the variable and can include a type modifier. For example, if *declarator* represents an array, the type of the pointer is modified to pointer to array.

You can declare a pointer to a structure, union, or enumeration type before you define the structure, union, or enumeration type. However, the definition must appear before the pointer can be used as an operand in an expression. You declare the pointer by using the structure or union tag (see Example 7 in this section). Such declarations are allowed because the compiler does not need to know the size of the structure or union to allocate space for the pointer variable.

Storage

The amount of storage required for an address and the meaning of the address depend on the implementation of the compiler. Pointers to different types are not guaranteed to have the same length.

In some implementations, you can use the special keywords **near**, **far**, and **huge** to modify the size of a pointer. Declarations using special keywords are described in ''Declarators with Special Keywords.'' For more information on the meaning and use of these keywords, see your compiler guide.

Variable Declarations

Example 1

This example declares a pointer variable named *message*. It points to a variable with **char** type.

```
char *message;
```

Example 2

Example 2 declares an array of pointers named *pointers*. The array has 10 elements; each element is a pointer to a variable with **int** type.

```
int *pointers[10];
```

Example 3

This example declares a pointer variable named *pointer*; it points to an array with 10 elements. Each element in this array has **int** type.

```
int (*pointer)[10];
```

Example 4

This example declares a pointer variable, *x*, to a constant value. The pointer may be modified to point to a different **int** value, but the value to which it points may not be modified.

```
int const *x;
```

Example 5

The variable *y* in Example 5 is declared as a constant pointer to an **int** value. The value it points to may be modified, but the pointer itself must always point to the same location: the address of *fixed_object*. Similarly, *z* is a constant pointer, but it is also declared to point to an **int** whose value will not be modified by the program. The additional specifier *volatile* indicates that although the value of the **const int** pointed to by *z* cannot be modified by the program, it could legitimately be modified by a process outside the program. The declaration of *w* specifies that the value pointed to will not be changed and that the program itself will not modify the pointer. However, some outside process could legitimately modify the pointer.

```
const int some_object = 5 ;
int other_object = 37;
int *const y = &fixed_object;
const volatile *const z = &some_object;
*const volatile w = &some_object;
```

Example 6

This example declares two pointer variables that point to the structure type *list*. This declaration can appear before the definition of the *list* structure type (see Example 7), as long as the *list* type definition has the same visibility as the declaration.

```
struct list *next, *previous;
```

Example 7

This example defines the variable *line* to have the structure type named *list*. The *list* structure type has three members: the first member is a pointer to a **char** value, the second is an **int** value, and the third is a pointer to another *list* structure.

```
struct list {
            char *token;
            int count;
            struct list *next;
    } line;
```

Example 8

This example declares the variable *record* to have the structure type *id*. Note that *pname* is declared as a pointer to another structure type named *name*. This declaration can appear before the *name* type is defined.

```
struct id {
     unsigned int id_no;
     struct name *pname;
    } record;
```

Example 9

In this example, the pointer variable *p* is declared, but the **void *** preceding the identifier *p* in the declaration means that *p* can be used later to point to any type object. The address of an **int** value is assigned to *p*, but no operations on the pointer itself are permitted unless it is explicitly converted to the type to which it points. Similarly, indirect operations on the object pointed to by *p* are not permitted unless *p* is converted to a specific type. Finally, a cast is used to convert *p* to a pointer to **int**, and *p* is then incremented.

```
int i;
void *p;         /* p declared as pointer to an object
                   whose type is not specified        */
p = &i;          /* address of integer i assigned to p
                   but type of p itself is still not
                   specified. An operation like p++
                   would not be permitted yet         */
 (int *)p++;     /* incrementing p permitted when the
                   cast converts it to pointer to int */
```

Function Declarations (Prototypes)

Syntax

[*sc-spec*] [*type-spec*] *declarator*([*formal-parameter-list*]) [, *declarator-list*]...;

A "function declaration," also called a "function prototype," establishes the name and return type of a function and may specify the types, formal parameter names, and number of arguments to the function. A function declaration does not define the function body. It simply makes information about the function known to the compiler. This information enables the compiler to check the types of the actual arguments in ensuing calls to the function.

If you do not provide a function prototype, the compiler constructs one from the first reference to the function it encounters, whether a call or a function definition. Whether such a prototype reflects the correct parameter types can only be assured if the function definition occurs in the same source file. If the definition occurs in a different module, argument mismatch errors may not be detected. Function definitions are described in detail in "Function Prototypes (Declarations)."

The *sc-spec* represents a storage-class specifier; it can be either **extern** or **static**. Storage-class specifiers are discussed in "Storage Classes."

The *type-spec* gives the function's return type, and *declarator* names the function. If you omit *type-spec* from a function declaration, the function is assumed to return a value of type **int**.

The *formal-parameter-list* is described in the next section.

The final *declarator-list* in the syntax line represents further declarations on the same line. These may be other functions returning values of the same type as the first function, or declarations of any variables whose type is the same as the first function's return type. Each such declaration must be separated from its predecessors and successors by a comma.

Formal Parameters

‘‘Formal parameters’’ describe the actual arguments that can be passed to a function. In a function declaration, the parameter declarations establish the number and types of the actual arguments. They may also include identifiers of the formal parameters. Though the parameters may be omitted from a function declaration, their inclusion is recommended, and they are mandatory in a true prototype. The extent of the information in the declaration influences the argument checking done on function calls that appear before the compiler has processed the function definition.

Note

> Identifiers used to name the formal parameters in the prototype declaration are descriptive only. They go out of scope at the end of the declaration. Therefore, they need not be identical to the identifiers used in the declaration portion of the function definition. Using the same names may enhance readability, but this use has no other significance.

Return Type

Functions can return values of any type except arrays and functions. Therefore, the *type-specifier* argument of a function declaration can specify any fundamental, structure, or union type. You can modify the function identifier with one or more asterisks (*) to declare a pointer return type.

Although functions cannot return arrays and functions, they can return pointers to arrays and functions. You can declare a function that returns a pointer to an array or function type by modifying the function identifier with asterisks (*), brackets ([ ]), and parentheses (( )). Such a function identifier is known as a ‘‘complex declarator.’’ Rules for forming and interpreting complex declarators are discussed in ‘‘Complex Declarators.’’

The List of Formal Parameters

All elements of the *formal-parameter-list* argument appearing within the parentheses following the function declarator are optional. The two following syntax variations illustrate the possibilities:

> [**void**]
> [**register**] [*type-spec*] [*declarator*[[, ...][,...]]]

If formal parameters are omitted from the function declaration, the parentheses should contain the keyword **void** to specify that no arguments will ever be passed to the function. If the parentheses are left entirely empty, no information is conveyed about whether arguments will be passed to the function and no checking of argument types is performed.

Note

> Empty parentheses in a function declaration or definition represent an obsolete form not recommended for new code. Functions accepting no arguments should be declared with the **void** keyword replacing the list of formal parameters. This use of **void** is interpreted by context, and is distinct from uses of **void** as a type specifier.

A declaration in the list of formal parameters can contain the **register** storage-class specifier, either alone or combined with a type specifier and an identifier. If **register** is not specified, the storage class is **auto**. The only explicit storage-class specifier permitted is **register**. If the parentheses contain only the **register** keyword, the formal parameter is considered to represent an unnamed **int** for which **register** storage is being requested.

If *type-spec* is included, it can specify the type name for any fundamental, structure, or union type (such as **int**). A *declarator* for a fundamental, structure, or union type is simply an identifier of a variable having that type.

The *declarator* for a pointer, array, or function can be formed by combining a type specifier, plus the appropriate modifier, with an identifier. Alternatively, an ''abstract declarator'' (that is, a declarator without a specified identifier) can be used. The section ''Type Names'' explains how to form and interpret abstract declarators.

A full, partial, or empty list of formal parameters can be declared. If the list contains at least one declarator, a variable number of parameters can be specified by ending the list with a comma followed by three periods (,...), referred to as the "ellipsis notation." A function is expected to have at least as many arguments as there are declarators or type specifiers preceding the last comma.

Note

> To maintain compatibility with previous versions, the compiler accepts a comma without trailing periods at the end of a declarator list to indicate a variable number of arguments. However, this is a Microsoft extension to the ANSI C standard. New code should use the comma followed by three periods. For information on enabling and disabling extensions, see your compiler guide.

One other special construction is permitted as a formal parameter: **void *** represents a pointer to an object of unspecified type. Thus, in a call, the pointer can refer to any type of object after you convert the pointer (for example, with a cast) to a pointer to the desired type. Note that before operations can be performed on the pointer or the object it addresses, the pointer must be explicitly converted. For more information on **void ***, see "Pointer Declarations."

Summary

Function prototypes are optional, but strongly recommended. If included, the only elements absolutely required are the name of the function, the opening and closing parentheses following the name, and the final semicolon. If no return type is included, as in the following example, the function is assumed to return an **int**:

```
/***** Obsolete form of function declaration *****/


minimal_declaration();      /* may or may not
                           accept arguments */
```

A full function prototype is the same as a function definition, except that instead of having a function "body," it is terminated by a semicolon (;) immediately following the closing parenthesis.

Any appropriate combination of elements is permitted among the parameter declarations, from no information (as in the obsolete form in the example above) to a full prototype of the function. If no prototype at all is given, a *de facto* prototype is constructed from information in the first reference to the function encountered in the source file.

Example 1

In this example, any information included in the formal parameter list is used to check actual arguments appearing in calls to the function that occur before the compiler has processed the function definition.

```
double func(void);       /* returns a double, but
                          * accepts no arguments
                         */
fun (void *);            /* takes a pointer to an
                          * unspecified type;
                          * returns an int
                         */
char *true(long, long);    /* takes two longs;
                          * returns pointer to char
                         */
new (register a, char *);  /* takes an int with request
                          * for register storage, and
                          * a pointer to char;
                          * returns an int
                         */
void go(int *[], char *b); /* takes an array of pointers
                          * to int using an abstract
                          * declarator, and a pointer
                          * to char; there is no return
                         */
void *tu(double v,...);         /* takes at least one double;
                          * other arguments may also be
                           * given; returns a pointer
                          * to an unspecified type
                         */
```

Example 2

This example is a prototype for a function named *add* that takes two **int** arguments, represented by the identifiers *num1* and *num2*, and returns an **int** value.

```
int add(int num1, int num2);
```

Example 3

This example declares a function named *calc* that returns a **double** value. The obsolete empty parentheses leave the issue of possible arguments to the function undefined.

```
double calc();
```

Example 4

This example is a prototype for a function named *strfind* that returns a pointer to **char**. The function accepts at least one argument, declared by the formal parameter *char *ptr*, to be a pointer to a **char** value. The formal parameter list has one entry and ends with a comma followed by three periods, indicating that the function may take more arguments.

```
char *strfind(char *ptr,...);
```

Example 5

This example declares a function with **void** return type (returning no value). The **void** keyword also replaces the list of formal parameters, so no arguments are expected for this function.

```
void draw(void);
```

Example 6

In this example, *sum* is declared as a function returning a pointer to an array of three **double** values. The *sum* function takes two **double** values as arguments.

```
double (*sum(double, double))[3];
```

Example 7

In this example, the function named *select* is declared to take no arguments and to return a pointer to a function. The pointer return value points to a function taking one **int** argument, represented by the identifier *number*, and returning an **int** value.

```
int (*select(void))(int number);
```

Example 8

In this example, the function *prt* is declared to take a pointer argument of any type and return an **int** value. A pointer to any type could be passed as an argument to *prt* without producing a type-mismatch warning.

```
int prt(void *);
```

Example 9

This example shows the declaration of an array, named *rainbow*, of an unspecified number of constant pointers to functions. Each of these takes at least one parameter of type **int**, as well as an unspecified number of other parameters. Each of the functions pointed to returns a **long** value.

```
long (*const rainbow[]) (int, ...) ;
```

Storage Classes

The “storage class” of a variable determines whether the item has a “local” or “global” lifetime. Variables with local lifetimes are allocated new storage each time execution control passes to the block in which they are defined. When execution control passes out of the block, the variables no longer have meaningful values.

An item with a global lifetime exists and has a value throughout the execution of the program. All functions have global lifetimes.

Although C defines only two types of storage classes, it provides the following four storage-class specifiers:

Table 4.5

Storage-Class Specifiers

Items declared with	Have a
auto	local lifetime
register	local lifetime
static	global lifetime
extern	global lifetime

The four storage-class specifiers have distinct meanings because they affect the visibility of functions and variables, as well as their storage class. The term “visibility” refers to the portion of the source program in which the variable or function can be referenced by name. An item with a global lifetime exists throughout the execution of the source program, but it may not be “visible” in all parts of the program. (Visibility and the related concept of lifetime are discussed in the chapter on “Program Structure.”)

The placement of variable and function declarations within source files also affects storage class and visibility. Declarations outside all function definitions are said to appear at the “external level”; declarations within function definitions appear at the “internal level.”

The exact meaning of each storage-class specifier depends on two factors:

- Whether the declaration appears at the external or internal level
- Whether the item being declared is a variable or a function

The sections that follow describe the meanings of storage-class specifiers in each kind of declaration and explain the default behavior when the storage-class specifier is omitted from a variable or function declaration.

Variable Declarations at the Global Level

In variable declarations at the global level (that is, outside all functions), you can use the **static** or **extern** storage-class specifier or omit the storage-class specifier entirely. You cannot use the **auto** and **register** storage-class specifiers at the global level.

Variable declarations at the global level are either *definitions* of variables (“defining declarations”), or *references* to variables defined elsewhere (“referencing declarations”).

A global variable declaration that also initializes the variable (implicitly or explicitly) is a defining declaration of the variable. A definition at the global level can take several forms:

- A variable that you declare with the **static** storage-class specifier. You can explicitly initialize the **static** variable with a constant expression, as described in “Initialization.” If you omit the initializer, the variable is initialized to 0 by default. For example, *static int k = 16;* and *static int k;* are both considered definitions of the variable *k*.
- A variable that you explicitly initialize at the global level. For example, *int j = 3;* is a definition of the variable *j*.

Once a variable is defined at the global level, it is visible throughout the rest of the source file in which it appears. The variable is not visible prior to its definition in the same source file. Also, it is not visible in other source files of the program, unless a referencing declaration makes it visible, as described later in this section.

You can define a variable at the global level only once within a source file. If you give the **static** storage-class specifier, you can define another variable with the same name and the **static** storage-class specifier in a different source file. Since each **static** definition is visible only within its own source file, no conflict occurs.

The **extern** storage-class specifier declares a *reference* to a variable defined elsewhere. You can use an **extern** declaration to make a definition in another source file visible, or to make a variable visible above its definition in the same source file. Once you have declared a reference to the variable at the global level, the variable is visible throughout the remainder of the source file in which the declared reference occurs.

Declarations that use the **extern** storage-class specifier cannot contain initializers, since these declarations refer to variables whose values are defined elsewhere.

For an **extern** reference to be valid, the variable it refers to must be defined once, and only once, at the global level. The definition can be in any of the source files that form the program.

One special case is not covered by the rules outlined above. You can omit both the storage-class specifier and the initializer from a variable declaration at the global level; for example, the declaration *int n;* is a valid global declaration. This declaration can have one of two different meanings, depending on the context:

1. If there is a global defining declaration of a variable with the same name elsewhere in the program, the current declaration is assumed to be a reference to the variable in the defining declaration, exactly as if the **extern** storage-class specifier had been used in the declaration.

2. If there is no global declaration of a variable with the same name elsewhere in the program, the declared variable is allocated storage at link time and initialized to 0. This kind of variable is known as a ''communal'' variable. If more than one such declaration appears in the program, storage is allocated for the largest size declared for the variable. For example, if a program contains two uninitialized declarations of *i* at the global level, *int i;* and *char i;*, storage space for an **int** value is allocated for *i* at link time.

Uninitialized variable declarations at the global level are not recommended for any file that might be placed in a library.

Example

The two source files in this example contain a total of three global declarations of *i*. Only one declaration contains an initialization; that declaration, *int i = 3;* , defines the global variable *i* with initial value 3. The **extern** declaration of *i* at the top of the first source file makes the global variable visible above its definition in the file. Without the **extern** declaration, the *main* function could not reference the global variable *i*. The **extern** declaration of *i* in the second source file also makes the global variable visible in that source file.

Assuming that the *printf* function is defined elsewhere in the program, all three functions perform the same task: they increase *i* and print it. The values 4, 5, and 6 are printed.

If the variable *i* had not been initialized, it would have been set to 0 automatically at link time. In this case, the values 1, 2, and 3 would have been printed.

Source File One

```
extern int i;                 /* reference to i,
                                 defined below */

main()
{
        i++;
        printf("%d\n", i);    /* i equals 4 */
        next();
}

int i = 3;                    /* definition of i */

next()
{
        i++;
        printf("%d\n", i);    /* i equals 5 */
        other();
}
```

Source File Two

```
extern int i;             /* reference to i in
                             first source file */

other()
{
        i++;
        printf("%d\n", i);   /* i equals 6 */
}
```

Variable Declarations at the Local Level

You can use any of the four storage-class specifiers for variable declarations at the local level. When you omit the storage-class specifier from such a declaration, the default storage class is **auto**.

The **auto** storage-class specifier declares a variable with a local lifetime. An **auto** variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed in ''Initialization.'' Since variables with **auto** storage class are not initialized automatically, you should either explicitly initialize them when you declare them, or assign them initial values in statements within the block. The values of uninitialized **auto** variables are undefined.

A **static auto** variable can be initialized with the address of any global or **static** item, but not with the address of another **auto** item, because the address of an **auto** item is not a constant.

The **register** storage-class specifier tells the compiler to give the variable storage in a register, if possible. Register storage usually speeds access time and reduces code size. Variables declared with **register** storage class have the same visibility as **auto** variables. The number of registers that can be used for variable storage is machine-dependent. If no registers are available when the compiler encounters a **register** declaration, the variable is given **auto** storage class and stored in memory. The compiler assigns register storage to variables in the order in which the declarations appear in the source file. Register storage, if available, is only guaranteed for **int** and pointer types that are the same size as an **int**.

A variable declared at the local level with the **static** storage-class specifier has a global lifetime but is visible only within the block in which it is declared. Unlike **auto** variables, **static** variables keep their values when the block is exited. You can initialize a **static** variable with a constant expression. A **static** variable is initialized only once, when program execution begins; it is *not* reinitialized each time the block is entered. If you do not explicitly initialize a **static** variable, it is initialized to 0 by default.

A variable declared with the **extern** storage-class specifier is a reference to a variable with the same name defined at the global level in any of the source files of the program. The local **extern** declaration is used to make the global-level variable definition visible within the block. Unless otherwise declared at the global level, a variable declared with the **extern** keyword is visible only in the block in which it is declared.

Example

In the following example, the variable *i* is defined at the global level with initial value 1. An **extern** declaration in the *main* function is used to declare a reference to the global-level *i*. The **static** variable *a* is initialized to 0 by default, since the initializer is omitted. The call to *printf* (assuming the *printf* function is defined elsewhere in the source program) prints the values 1, 0, 0, and 0.

In the *other* function, the address of the global variable *i* is used to initialize the **static** pointer variable *external_i*. This works because the global variable has **static** lifetime, meaning its address will always be the same. Next, the variable *i* is redefined as a local variable with initial value 16. This redefinition does not affect the value of the global-level *i*, which is hidden by the use of its name for the local variable. The value of the global *i* is now accessible only indirectly within this block, through the pointer *external_i*. Attempting to assign the address of the **auto** variable *i* to a pointer would not work, since it may be different each time the block is entered. The variable *a* is declared as a **static** variable and initialized to 2. This *a* does not conflict with the *a* in *main*, since **static** variables at the local level are visible only within the block in which they are declared.

The variable *a* is increased by 2, giving 4 as the result. If the *other* function were called again in the same program, the initial value of *a* would be 4, since local **static** variables keep their values when the program exits and then re-enters the block in which they are declared.

```
int i = 1;

main()
{       /* reference to i, defined above: */
        extern int i;

        /* initial value is zero; a is
           visible only within main: */
        static int a;

        /* b is stored in a register, if possible: */
        register int b = 0;

        /* default storage class is auto: */
        int c = 0;

        /* values printed are 1, 0, 0, 0: */
        printf("%d\n%d\n%d\n%d\n", i, a, b, c);
        other();
}

other()
{
       /* address of global i assigned to pointer variable */
       static int *external_i = &i;

        /* i is redefined; global i no longer visible: */
        int i = 16;

        /* this a is visible only within other: */
        static int a = 2;

        a += 2;
        /* values printed are 16, 4, and 1: */
        printf("%d\n%d\n%d\n", i, a, *external_i);
}
```

Function Declarations at the Global and Local Levels

You can use either the **static** or the **extern** storage-class specifier in function declarations. Functions always have global lifetimes.

The visibility rules for functions vary slightly from the rules for variables, as follows:

- A function declared to be **static** is visible only within the source file in which it is defined. Functions in the same source file can call the **static** function, but functions in other source files cannot. You can declare another **static** function with the same name in a different source file without conflict.

- Functions declared as **extern** are visible throughout all the source files that make up the program (unless you later redeclare such a function as **static**). Any function can call an **extern** function.
- Function declarations that omit the storage-class specifier are **extern** by default.

Note

A Microsoft extension to the ANSI C standard provides that function declarations at the local level have the same meaning as function declarations at the global level. This means that a function is visible from its point of declaration through the rest of the source file.

Initialization

Syntax

declarator = initializer

You can set a variable to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable. An equal sign (=) precedes the initializer.

You can initialize variables of any type, provided that you obey the following rules:

- Declarations that use the **extern** storage-class specifier cannot include initializers.
- Variables declared at the global level can be initialized. If you do not explicitly initialize a variable at the global level, it is initialized to 0 by default.
- A constant expression can be used to initialize any variable declared with the **static** storage-class specifier. Variables declared to be **static** are initialized when program execution begins. If you do not explicitly initialize a **static** variable, it is initialized to 0 by default.
- Variables declared with the **auto** and **register** storage-class specifiers are initialized each time execution control passes to the block in which they are declared. If you omit an initializer from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined.
- Aggregate types with **auto** storage class (arrays, structures, and unions) cannot be initialized. Only **static** aggregates and aggregates declared at the global level can be initialized.
- The initial values for global variable declarations and for all **static** variables, whether global or local, must be constant expressions. You can use either constant or variable values to initialize **auto** and **register** variables.

The sections that follow describe how to initialize variables of fundamental, pointer, and aggregate types.

Fundamental and Pointer Types

Syntax

declarator = expression

The value of *expression* is assigned to the variable. The conversion rules for assignment apply.

A locally declared static variable can only be initialized with a constant value. Since the address of any globally declared or static variable is constant, it may be used to initialize an local declared **static** pointer variable. However, the address of an **auto** variable cannot be used as an initializer because it may be different for each execution of the block.

Example 1

In this example, *x* is initialized to the constant expression 10.

```
int x = 10;
```

Example 2

In this example, the pointer *px* is initialized to 0, producing a "null" pointer.

```
register int *px = 0;
```

Example 3

This example uses a constant expression to initialize *c* to a constant value that cannot be modified.

```
const int c = (3 * 1024);
```

Example 4

This example initializes the pointer *b* with the address of another variable, *x*. The pointer *a* is initialized with the address of a variable named *z*. However, since it is specified to be a **const**, the variable *a* can only be initialized, never modified. It always points to the same location.

```
int *b = &x;
int *const a = &z;
```

Example 5

The global variable *GLOBAL* is declared in Example 5 at the global level, so it has global lifetime. The local variable *LOCAL* has **auto** storage class and only has an address during the execution of the function in which it is declared. Therefore, attempting to initialize the **static** pointer variable *lp* with the address of *LOCAL* is not permitted. The **static** pointer variable *gp* can be initialized to the address of *GLOBAL* because that address is always the same. Similarly, **rp* can be initialized because *rp* is a local variable and can have a nonconstant initializer. Each time the block is entered, *LOCAL* will have a new address, which will then be assigned to *rp*.

```
int GLOBAL ;

int function(void)
  {
     int LOCAL ;
     static int *lp = &LOCAL;   /* Illegal declaration */
     static int *gp = &GLOBAL;  /* Legal declaration   */
     register int *rp = &LOCAL; /* Legal declaration   */
  }
```

Aggregate Types

Syntax

declarator = {*initializer-list*}

The *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant expression or an initializer list. Therefore, an initializer list enclosed in braces can appear within another initializer list. This form is useful for initializing aggregate members of an aggregate type, as shown in the in this section.

For each *initializer-list*, the values of the constant expressions are assigned, in order, to the corresponding members of the aggregate variable. When a union is initialized, *initializer-list* must be a single constant expression. The value of the constant expression is assigned to the first member of the union.

If *initializer-list* has fewer values than an aggregate type, the remaining members or elements of the aggregate type are initialized to 0. If *initializer-list* has more values than an aggregate type, an error results.

These rules apply to each embedded initializer list, as well as to the aggregate as a whole.

The following example, declares *P* as a 4-by-3 array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row:

```
int P[4][3] = {
        { 1, 1, 1 },
        { 2, 2, 2 },
        { 3, 3, 3,},
        { 4, 4, 4,},
};
```

Note that the initializer list for the third and fourth rows contains commas after the last constant expression. The last initializer list *({4,4,4,})* is also followed by a comma. These extra commas are permitted but not required; only commas that separate constant expressions from one another, and those that separate one initializer list from another, are required.

If there is no embedded initializer list for an aggregate member, values are simply assigned, in order, to each member of the subaggregate. Therefore, the initialization in the previous example is equivalent to the following:

```
int P[4][3] = {
        1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list.

When you initialize an aggregate variable, you must be careful to use braces and initializer lists properly. The following example illustrates the compiler's interpretation of braces in more detail:

```
typedef struct {
    int n1, n2, n3;
} triplet;

triplet nlist[2][3] = {
    { {  1, 2, 3 }, {  4, 5, 6 }, {  7, 8, 9 } },  /* Line 1 */
    { { 10,11,12 }, { 13,14,15 }, { 16,17,18 } }   /* Line 2 */
};
```

In this example, *nlist* is declared as a 2-by-3 array of structures, each structure having three members. Line 1 of the initialization assigns values to the first row of *nlist*, as follows:

1. The first left brace on Line 1 signals the compiler that the first aggregate member of *nlist* (that is, *nlist[0]*) is initializing.
2. The second left brace indicates that the first aggregate member of *nlist[0]* (that is, the structure at *nlist[0][0]*) is initializing.
3. The first right brace ends initialization of the structure *nlist[0][0]*; the next left brace starts initializing *nlist[0][1]*.
4. The process continues until the end of the line, where the closing right brace ends initialization of *nlist[0]*.

Line 2 assigns values to the second row of *nlist* in a similar way.

Note that the outer sets of braces enclosing the initializers on lines 1 and 2 are required. The following construction, which omits the outer braces, would cause an error:

```
triplet nlist[2][3] = {
      {  1, 2, 3 },{  4, 5, 6 },{  7, 8, 9 },   /* Line 1 */
      { 10,11,12 },{ 13,14,15 },{ 16,17,18 }    /* Line 2 */
};
```

In this construction, the first left brace on line 1 initializes *nlist[0]*, which is an array of three structures. The values 1, 2, and 3 are assigned to the three members of the first structure. When the next right brace is encountered (after the value 3), initialization of *nlist[0]* is complete, and the two remaining structures in the three-structure array are automatically initialized to 0. Similarly, *{ 4,5,6 }* initializes the first structure in the second row of *nlist*. The remaining two structures of *nlist[1]* are set to 0. When the compiler encounters the next initializer list (*{ 7,8,9 }*), it tries to initialize *nlist[2]*. Since *nlist* has only two rows, this attempt causes an error.

Example 1

In this example, the three **int** members of *x* are initialized to 1, 2, and 3, respectively. The three elements in the first row of *m* are initialized to 4.0; the elements of the remaining row of *m* are initialized to 0.0 by default.

```
struct list {
        int i, j, k;
        float m[2][3];
        } x = {
                1,
                2,
                3,
                {4.0, 4.0, 4.0}
        };
```

Example 2

In this example, the union variable *y* is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list *{’1’}* assigns values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized to the character *1*, and the remaining two elements in the row are initialized to the value zero by default. Similarly, the first element of the second row of *x* is initialized to the character *4*, and the remaining two elements in the row are initialized to the value 0.

```
union
 {
        char x[2][3];
        int i, j, k;
        } y = { {
                 {’1’},
                 {’4’} }
        };
```

String Initializers

Syntax

declarators = "*characters*"

You can initialize an array of characters with a string literal. The following example, initializes *code* as a four-element array of characters. The fourth element is the null character, which terminates all string literals.

```
char code[ ] = "abc";
```

If you specify the array size and the string is longer than the specified array size, the extra characters are simply ignored. For example, the following declaration initializes *code* as a three-element character array:

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to *code*. The character *d* and the string-terminating null character are discarded. Beware that this creates an unterminated string (that is, one without a 0 value to mark its end) and generates a diagnostic message indicating the condition.

If the string is shorter than the specified array size, the remaining elements of the array are initialized to 0 values.

Type Declarations

A type declaration defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type. You can use the name of a declared type in variable or function declarations to refer to that type. This is useful if many variables and functions have the same type.

A **typedef** declaration defines a type specifier for a type. You can use **typedef** declarations to construct shorter or more meaningful names for types already defined by C or for types that you have declared.

Structure, Union, and Enumeration Types

Declarations of structure, union, and enumeration types have the same general form as variable declarations of those types. However, type declarations and variable declarations differ in the following ways:

- In type declarations the variable identifier is omitted, since no variable is declared.
- In type declarations *tag* is required; it names the structure, union, or enumeration type.
- The *member-declaration-list* or *enum-list* defining the type must appear in the type declaration; the abbreviated form of variable declarations, in which *tag* refers to a type defined elsewhere, is not legal for type declarations.

Example 1

This example declares an enumeration type named *status*. The name of the type can be used in declarations of enumeration variables. The identifier *loss* is explicitly set to -1. Both *bye* and *tie* are associated with the value 0, and *win* is given the value 1.

```
enum status {
        loss = -1,
        bye,
        tie = 0,
        win
        };
```

Example 2

This example declares a structure type named *student.* A declaration such as *struct student employee;* can be used to define a structure variable with *student* type.

```
struct student {
        char name[20];
        int id, class;
        };
```

Using typedef Declarations

Syntax

typedef *type-specifier declarator* [**,** *declarator*]...**;**

A **typedef** declaration is analogous to a variable declaration except that the **typedef** keyword replaces a storage-class specifier. A **typedef** declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

Note that a **typedef** declaration does not create types. It creates synonyms for existing types, or names for types that could be specified in other ways. When a **typedef** name is used as a type specifier, it can be combined with certain type specifiers, but not others. Acceptable modifiers include **const** and **volatile**. In some implementations there are additional special keywords that can be used to modify a **typedef**.

You can declare any type with **typedef**, including pointer, function, and array types. You can declare a **typedef** name for a pointer to a structure or union type before you define the structure or union type, as long as the definition has the same visibility as the declaration.

Example 1

This example declares *WHOLE* to be a synonym for **int**. Note that *WHOLE* could now be used in a variable declaration such as *WHOLE i;* or *const WHOLE i;*. However, the declaration *long WHOLE i;* would be illegal.

```
typedef int WHOLE;
```

Example 2

This example declares *GROUP* as a structure type with three members. Since a structure tag, *club*, is also specified, either the **typedef** name (*GROUP*) or the structure tag can be used in declarations.

```
typedef struct club {
        char name[30];
        int size, year;
        } GROUP ;
```

Example 3

This example uses the previous **typedef** name to declare a pointer type. The type *PG* is declared as a pointer to the *GROUP* type, which in turn is defined as a structure type.

```
typedef GROUP *PG;
```

Example 4

Example 4 provides the type *DRAWF* for a function returning no value and taking two **int** arguments. This means, for example, that the declaration *DRAWF box;* is equivalent to the declaration *void box(int, int);*.

```
typedef void DRAWF(int, int);
```

Type Names

A ‘‘type name’’ specifies a particular data type. In addition to ordinary variable declarations and defined-type declarations, type names are used in three other contexts: in the formal-parameter lists of function declarations, in type casts, and in **sizeof** operations. Formal-parameter lists are discussed in ‘‘Function Declarations.’’ Type casts and **sizeof** operations are discussed in Chapter 5 in the sections entitled ‘‘Type Conversions’’ and ‘‘C Operators’’, respectively.

The type names for fundamental, enumeration, structure, and union types are simply the type specifiers for those types.

A type name for a pointer, array, or function type has the following form:

type-specifier abstract-declarator

An *abstract-declarator* is a declarator without an identifier, consisting of one or more pointer, array, or function modifiers. The pointer modifier (*) always precedes the identifier in a declarator; array ([ ]) and function (( )) modifiers follow the identifier. Knowing this, you can determine where the identifier would appear in an abstract declarator and interpret the declarator accordingly.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations.

Note

The abstract declarator consisting of a set of empty parentheses, ( ), is not allowed because it is ambiguous. It is impossible to determine whether the implied identifier belongs inside the parentheses (in which case it is an unmodified type) or before the parentheses (in which case it is a function type).

The type specifiers established by **typedef** declarations also qualify as type names.

Example 1

This example gives the type name for ‘‘pointer to **long**’’ type.

```
long *
```

Example 2

Examples 2 and 3 show how parentheses modify complex abstract declarators. Example 2 gives the type name for a pointer to an array of five **int** values.

```
int (*)[5]
```

Example 3

Example 3 specifies a pointer to a function taking no arguments and returning an **int** value.

```
int (*)(void)
```

Chapter 5

Expressions and Assignments

Introduction 5-1
    Constants 5-2
    Identifiers 5-2
    Strings 5-3
    Function Calls 5-3
    Subscript Expressions 5-4
    Member-Selection Expressions 5-7
    Expressions with Operators 5-9
    Expressions in Parentheses 5-10
    Type-Cast Expressions 5-10
    Constant Expressions 5-11
    Side Effects 5-12
    Sequence Points 5-12

C Operators 5-14
    Usual Arithmetic Conversions 5-15
    Complement and Unary Plus Operators 5-16
    Indirection and Address-of Operators 5-17
    The sizeof Operator 5-19
    Multiplicative Operators 5-20
    Additive Operators 5-22
    Shift Operators 5-24
    Relational Operators 5-25
    Bitwise Operators 5-27
    Logical Operators 5-28
    Sequential-Evaluation Operator 5-29
    Conditional Operator 5-30

Assignment Operators 5-32
    Lvalue Expressions 5-33
    Unary Increment and Decrement 5-34
    Simple Assignment 5-35
    Compound Assignment 5-36

Precedence and Order of Evaluation 5-37

Type Conversions 5-41
    Assignment Conversions 5-41

Type-Cast Conversions 5-50
Operator Conversions 5-50
Function-Call Conversions 5-50

Introduction

This chapter describes how to form expressions and make assignments in the C language. An ‘‘expression’’ is a combination of operands and operators that yields (‘‘expresses’’) a single value.

An ‘‘operand’’ is a constant or variable value that is manipulated in the expression. Each operand of an expression is also an expression, since it represents a single value. When an expression is evaluated, the resulting value depends on the relative precedence of operators in the expression and on ‘‘sequence points’’ and ‘‘side effects,’’ if any. The precedence of operators determines how operands are grouped for evaluation. Side effects are changes caused by the evaluation of an expression. In an expression with side effects, the evaluation of one operand can affect the value of another. With some operators, the order in which operands are evaluated also affects the result of the expression. The section entitled ‘‘C Operands’’ describes the formats and evaluation rules for C operands, including discussions of side effects and sequence points.

‘‘Operators’’ specify how the operand or operands of the expression are manipulated. C operators are described later in this chapter in the section entitled ‘‘C Operators’’.

In C, assignments are considered expressions because an assignment yields a value. Its value is the value being assigned. In addition to the simple-assignment operator (=), C offers complex-assignment operators that both transform and assign their operands. Assignment operators are described later in this chapter in the section entitled ‘‘Assignment Operators’’.

The value represented by each operand in an expression has a type that may be converted to a different type in certain contexts. Type conversions occur in assignments, type casts, function calls, and operations. (The section entitled ‘‘Precedence and Order of Evaluation’’ gives the precedence rules for C operators; side effects are discussed in the ‘‘Introduction’’ section of this chapter under ‘‘Side Effects’’; and type conversions are covered in the section entitled ‘‘Type Conversions’’ all elsewhere in this chapter.)

Operands in C include constants, identifiers, strings, function calls, subscript expressions, member-selection expressions, or more complex expressions formed by combining operands with operators or enclosing operands in parentheses. Any operand that yields a constant value is called a ‘‘constant expression.’’

Every operand has a type. The following sections discuss the type of value each kind of operand represents. An operand can be ‘‘cast’’ (or temporarily converted) from its original type to another type by means of a ''type-cast'' operation. A type-cast expression can also form an operand of an expression.

Constants

A constant operand has the value and type of the constant value it represents. A character constant has **int** type. An integer constant has **int**, **long**, **unsigned int**, or **unsigned long** type, depending on the integer's size and how the value is specified. Floating-point constants always have **double** type. String literals are considered arrays of characters and are discussed earlier in this section under the subheading ''Strings''.

Identifiers

An ''identifier'' names a variable or function. Every identifier has a type that is established when the identifier is declared. The value of an identifier depends on its type, as follows:

- Identifiers of integral and floating types represent values of the corresponding type.
- An identifier of **enum** type represents one constant value among a set of constant values. The value of the identifier is the constant value. Its type is **int**, by definition of the **enum** type.
- An identifier of **struct** or **union** type represents a value of the specified **struct** or **union** type.
- An identifier declared as a pointer represents a pointer to a value of the type specified in the pointer's declaration.
- An identifier declared as an array represents a pointer whose value is the address of the first array element. The pointer addresses the type of the array elements. For if *series* is declared to be a 10-element integer array, the identifier *series* represents the address of the array, and the subscript expression *series[5]* refers to an integer value which is the sixth element of *series*. Subscript expressions are discussed later in this section under the subheading ''Subscript Expressions''. The address of an array does not change during program execution, although the values of the individual elements can change. The pointer value represented by an array identifier is not a variable, so an array identifier cannot form the left-hand operand of an assignment operation.

- An identifier declared as a function represents a pointer whose value is the address of the function. The pointer addresses a function returning a value of a specified type. The address of a function does not change during program execution; only the return value varies. Thus, function identifiers cannot be left-hand operands in assignment operations.

Strings

Syntax

"*string*" ["*string*"]

A ''string literal'' is a character or sequence of adjacent characters enclosed in double quotation marks. Two or more adjacent string literals separated only by white space are concatenated into a single string literal. A string literal is stored as an array of elements with **char** type and initialized with the quoted sequence of characters. The string literal is represented by a pointer whose value is the address of the first array element. The address of the string's first element is a constant, so the value represented by a string expression is a constant.

Since string literals are effectively pointers, they can be used in the same contexts as pointers, and have the same restrictions as pointers. However, since they are not variables, neither string literals nor any of their elements can be the left-hand operand in an assignment operation.

The last character of a string is always the null character. Though the null character is not visible in the string expression, it is added automatically as the last element when the string is stored. For example, the string *"abc"* actually has four characters rather than three.

Function Calls

Syntax

expression ([*expression-list*])

A ''function call'' consists of an *expression* followed by an optional *expression-list* in parentheses, where

- *expression* must evaluate to a function address (for example, a function identifier), and

- *expression-list* is a list of expressions (separated by commas) whose values (the "actual arguments") are passed to the function. The *expression-list* argument can be empty.

A function-call expression has the value and type of the function's return value. If the function's return type is **void** (that is, the function has been declared never to return a value), the function-call expression also has **void** type. If the called function returns control without executing a **return** statement, the value of the function-call expression is undefined. (See the chapter on "Functions," for more information about function calls.)

Subscript Expressions

Syntax

expression1 [*expression2*]

A subscript expression represents the value at the address that is *expression2* positions beyond *expression1*. Usually, the value represented by *expression1* is a pointer value, such as an array identifier, and *expression2* is an integral value. However, all that is required syntactically is that one of the expressions be of pointer type and the other be of integral type. Thus the integral value could be in the *expression1* position and the pointer value could be in the brackets in the *expression2*, or "subscript," position. Whatever the order of values, *expression2* must be enclosed in brackets ([ ]).

Subscript expressions are generally used to refer to array elements, but you can apply a subscript to any pointer.

Unidimensional-Array References

The subscript expression is evaluated by adding the integral value to the pointer value, then applying the indirection operator (*****) to the result. (For a discussion of the indirection operator see the "Indirection and Address-of Operators" subsection of the section entitled "C Operators" later in this chapter.) In effect, for a one-dimensional array, the following four expressions are equivalent, assuming that *a* is a pointer and *b* is an integer:

```
a[b]
*(a + b)
*(b + a)
b[a]
```

According to the conversion rules for the addition operator (see "Additive Operators" in the "C Operators" section), the integral value is converted to an address offset by multiplying it by the length of the type addressed by the pointer.

For example, suppose the identifier *line* refers to an array of **int** values. The following procedure is used to evaluate the subscript expression *line*[*i*]:

1. The integer value *i* is multiplied by the number of bytes defined as the length of an **int** item. The converted value of *i* represents *i* **int** positions.
2. This converted value is added to the original pointer value (*line*) to yield an address that is offset *i* **int** positions from *line.*
3. The indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, *line*[*i*]).

Note

The following subscript expression represents the value of the first element of *line*, since the offset from the address represented by *line* is 0:

```
line[0]
```

Similarly, an expression such as the following refers to the element offset five positions from *line* or the sixth element of the array:

```
line[5]
```

Multidimensional-Array Reference

A subscript expression can be subscripted, as follows:

expression1 [*expression2*] [*expression3*]...

Subscript expressions associate from left to right. The left-most subscript expression, *expression1*[*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms a pointer expression; then *expression3* is added to this pointer expression to form a new pointer expression, and so on until the last subscript expression has been added. The indirection operator (*) is applied after the last subscripted expression is evaluated, unless the final pointer value addresses an array type (see Example 3).

Expressions with multiple subscripts refer to elements of ‘‘multidimensional arrays.’’ A multidimensional array is an array whose elements are arrays. For example, the first element of a three-dimensional array is an array with two dimensions.

For the following examples, an array named *prop* is declared with three elements, each of which is a 4-by-6 array of **int** values.

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];
```

Example 1

This example shows how to refer to the second individual **int** element of *prop*. Arrays are stored by row, so the last subscript varies the most quickly; the expression *prop[0][0][2]* refers to the next (third) element of the array, and so on.

```
i = prop[0][0][1];
```

Example 2

This example shows a more complex reference to an individual element of *prop*. The expression is evaluated as follows:

1. The first subscript, 2, is multiplied by the size of a 4-by-6 **int** array and added to the pointer value *prop*. The result points to the third 4-by-6 array of *prop*.

2. The second subscript, *1*, is multiplied by the size of the 6-element **int** array and added to the address represented by *prop[2]*.

3. Each element of the 6-element array is an **int** value, so the final subscript, *3*, is multiplied by the size of an **int** before it is added to *prop[2][1]*. The resulting pointer addresses the fourth element of the 6-element array.

4. The indirection operator is applied to the pointer value. The result is the **int** element at that address.

```
i = prop[2][1][3];
```

Example 3

Examples 3 and 4 show cases where the indirection operator is not applied.

In Example 3, the expression *prop[2][1]* is a valid reference to the three-dimensional array *prop*; it refers to a 6-element array (declared above Example 1). Since the pointer value addresses an array, the indirection operator is not applied.

```
ip = prop[2][1];
```

Example 4

As in example 3, the result of the expression *prop[2]* in Example 4 is a pointer value addressing a two-dimensional array.

```
ipp = prop[2];
```

Member-Selection Expressions

Syntax

expression.identifier
expression->*identifier*

A "member-selection expression" refers to members of structures and unions. Such an expression has the value and type of the selected member. As shown in the syntax line, a member-selection expression can have one of the two following forms:

1. In the first form, *expression.identifier*, *expression* represents a value of **struct** or **union** type, and *identifier* names a member of the specified structure or union.
2. In the second form, *expression–>identifier*, *expression* represents a pointer to a structure or union, and *identifier* names a member of the specified structure or union.

The two forms of member-selection expressions have similar effects. In fact, an expression involving the pointer selection operator (–>) is a shorthand version of an expression using the period (.) if the expression before the period consists of the indirection operator (*) applied to a pointer value. (See the ‘‘Indirection and Address-of Operators’’ section in the ‘‘C Operators’’ section.) Therefore,

expression->*identifier*

is equivalent to

(**expression*).*identifier*

when *expression* is a pointer value.

Examples 1 through 3 refer to the following structure declaration:

```
struct pair {
        int a;
        int b;
        struct pair *sp;
        } item, list[10];
```

Example 1

In this example, the address of the *item* structure is assigned to the *sp* member of the structure. This means that *item* contains a pointer to itself.

```
item.sp = &item;
```

Example 2

In this example, the pointer expression *item.sp* is used with the pointer selection operator (–>) to assign a value to the member *a*.

```
(item.sp)->a = 24;
```

Example 3

This example shows how to select an individual structure member from an array of structures.

```
list[8].b = 12;
```

Expressions with Operators

Expressions with operators can be "unary," "binary," or "ternary" expressions. A unary expression consists of either a unary operator ("unop") prepended to an operand, or the **sizeof** keyword followed by an *expression*. The *expression* can be either the name of a variable or a cast expression. If *expression* is a cast expression it must be enclosed in parentheses.

unop operand
sizeof *expression*

A binary expression consists of two operands joined by a binary operator ("binop"):

operand binop operand

A ternary expression consists of three operands joined by the ternary operator (**? :**):

operand **?** *operand* **:** *operand*

The section later in this chapter entitled "C Operators" describes the operators used in unary, binary, and ternary expressions.

Expressions with operators also include assignment expressions that use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (--) operators; the binary assignment operators are the simple-assignment operator (=) and the compound-assignment operators (referred to as "compound-assign-ops"). Each compound-assignment operator is a combination of another binary operator with the simple-assignment operator. Assignment expressions have the following forms:

operand++
operand– –
++operand
– –operand
operand = operand
operand compound-assign-op operand

The section later in this chapter entitled ‘‘Assignment Operators’’ describes the assignment operators in detail.

Expressions in Parentheses

You can enclose any operand in parentheses without changing the type or value of the enclosed expression. For example, in the the following expression, the parentheses around *10 + 5* mean that the value of *10 + 5* is the left operand of the division (/) operator.

```
(10 + 5) / 5
```

The result of *(10 + 5) / 5* is 3. Without the parentheses, *10 + 5 / 5* would evaluate to 11.

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation in all cases. Exceptions resulting from ‘‘side effects’’ are discussed in the section entitled ‘‘Precedence and Order of Evaluation’’ later in this chapter.

Type-Cast Expressions

A type cast provides a method for explicit conversion of the type of an object in a specific situation. Type-cast expressions have the following form:

(*type-name*) *operand*

Casts can be used to convert objects of any scalar type to or from any other scalar type. Explicit type casts are constrained by the same rules that determine the effects of implicit conversions, discussed in ‘‘Assignment Conversions.’’ Additional restraints on casts may result from the actual sizes or representation of specific types on specific implementations. Representation is discussed in the ‘‘Declarations’’ chapter. For information on actual sizes of integral types and pointers, see your compiler guide.

Any object may be cast to **void** type. However, if the *type-name* in a type-cast expression is not **void**, then *operand* cannot be a **void** expression. Any expression can be cast to **void**, but an expression of type **void** cannot be cast to any other type. For example, a function with **void** return type cannot have its return cast to another type. Note that a **void** * expression has a type pointer to **void**, not type **void**. If an object is cast to **void** type, the resulting expression cannot be assigned to any item. Similarly, a type-cast object is not an acceptable lvalue, so no assignment can be made to a type-cast object. See the section on ''Lvalue Expressions'' in the section entitled ''Assignment Operators'' for a discussion of Lvalues, and the section on ''Type-Cast Conversions'' in the section entitled ''Type Conversions'' for a discussion of type conversions later in this chapter.

Constant Expressions

A constant expression is any expression that evaluates to a constant. The operands of a constant expression can be integer constants, character constants, floating type constants, enumeration constants, type casts, **sizeof** expressions, and other constant expressions. You can use operators to combine and modify operands as described in the section entitled ''Expressions with Operators'' with the following restrictions:

- You cannot use assignment operators (see ''Assignment Operators'' later in this chapter) or the binary sequential-evaluation operator (,) in constant expressions.

- You can use the unary address-of operator (**&**) only in certain initializations (as described in the last paragraph of this section).

Constant expressions used in preprocessor directives are subject to additional restrictions. Consequently, they are known as ''restricted constant expressions.'' A restricted constant expression cannot contain **sizeof** expressions, enumeration constants, type casts to any type, or floating-type constants. It can, however, contain the special constant expression **defined**(*identifier*).

Constant expressions involving floating constants, casts to nonarithmetic types, and address-of expressions can only appear in initializers. The unary address-of operator (**&**) can only be applied to variables with fundamental, structure, or union types that are declared at the global level, or to subscripted array references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address expression.

Side Effects

''Side effects'' occur whenever the value of a variable is changed by expression evaluation. All assignment operations have side effects. Function calls may also have side effects if they change the value of an externally visible item, either by direct assignment or by indirect assignment through a pointer.

The order of evaluation of expressions is defined by the specific implementation, except when the language guarantees a particular order of evaluation (as outlined in the section entitled ''Precedence and Order of Evaluation'' later in this chapter).

For example, side effects occur in the following function call:

```
add (i + 1, i = j + 2)
```

The arguments of a function call can be evaluated in any order. The expression $i + 1$ can be evaluated before $i = j + 2$, or $i = j + 2$ can be evaluated before $i + 1$. The result is different in each case.

Sequence Points

Expressions involving assignment, unary ''increment,'' unary ''decrement,'' or calling a function may have consequences incidental to their evaluation (side effects). When a ''sequence point'' is reached, everything preceding the sequence point, including any side effects, is guaranteed to have been evaluated before evaluation begins on anything following the sequence point.

Certain operators act as sequence points, including the following:

- The logical-AND operator (**&&**)
- The logical-OR operator (**||**)
- The ternary operator (**?:**)
- The sequential-evaluation operator (**,**)
- The function-call operator (that is, the parentheses following a function name)

Other sequence points include

- the end of a full expression (that is, an expression that is not part of another expression)
- any initializer
- an expression in an expression statement
- the control expressions in selection statements (**if** or **switch**) and iteration statements (**do**, **while**, or **for**)
- the expression in a **return** statement

C Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator). Assignment operators include both unary or binary operators; the section entitled ‘‘Assignment Operators’’ describes the assignment operators.

Unary operators appear before their operand and associate from right to left. C includes the following unary operators:

- ~ !	Negation and complement operators
*** &**	Indirection and address-of operators
sizeof	Size operator
+	Unary plus operator

Binary operators associate from left to right. C provides the following binary operators:

* / %	Multiplicative operators
+ −	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
,	Sequential-evaluation operator

C has one ternary operator: the conditional operator (**? :**). It associates from right to left.

Usual Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating types. These conversions are known as ''arithmetic conversions'' because they apply to the types of values ordinarily used in arithmetic.

The arithmetic conversions summarized in this section are called ''usual arithmetic conversions.'' The discussion of each operator in the following sections specifies whether or not the operator performs the usual arithmetic conversions. It also specifies the additional conversions, if any, the operator performs. This is not a precedence order. It is an outline of an algorithm that is applied to each binary operator in the expression.

The section entitled ''Type Conversions'' outlines the specific path of each type of conversion. In determining which conversions will actually take place, the following algorithm is applied to each binary operation in the expression:

1. Any operands of **float** type are converted to **double** type.
2. If one operand has **long double** type, the other operand is converted to **long double** type.
3. If one operand has **double** type, the other operand is converted to **double** type.
4. Any operands of **char** or **short** type are converted to **int** type.
5. Any operands of **unsigned char** or **unsigned short** type are converted to **unsigned int** type.
6. If one operand is of **unsigned long** type, the other operand is converted to **unsigned long** type.
7. If one operand is of **long** type, the other operand is converted to **long** type.
8. If one operand is of **unsigned int** type, the other operand is converted to **unsigned int** type.

The following example illustrates the application of the preceding algorithm:

```
long l;
unsigned char uc;
int i;
f( l + uc * i);
```

The preceding example would be converted as follows:

1. *uc* is converted to an **unsigned int** (step 5).

2. *i* is converted to an **unsigned int** (step 8). The multiplication is performed and the result is an **unsigned int**.

3. *uc* * *i* is converted to a **long** (step 7).

The addition is performed and the result is type **long.**

Complement and Unary Plus Operators

The C complement operators are discussed in the following list:

-	The arithmetic-negation operator produces the negative (two's complement) of its operand. The operand must be an integral or floating value. This operator performs the usual arithmetic conversions.
~	The bitwise-complement operator produces the bitwise complement of its operand. The operand must be of integral type. This operator performs usual arithmetic conversions; the result has the type of the operand after conversion.
!	The logical-NOT operator produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has **int** type. The operand must be an integral, floating, or pointer value.
+	The unary plus operator preceding a parenthesized expression forces the grouping of the enclosed operations. It is used with expressions involving more than one associative or commutative binary operator.

Note

> The unary plus operator (+) is implemented syntactically in Microsoft C, but has no semantics of any type associated with it.

Example 1

In this example, the new value of *x* is the negative of 987, or –987.

```
short x = 987;
    x = -x;
```

Example 2

In this example, the new value assigned to *y* is the one's complement of the unsigned value 0xaaaa, or 0x5555.

```
unsigned short y = 0xaaaa;
    y = ~y;
```

Example 3

In this example, if *x* is greater than or equal to *y*, the result of the expression is 1 (true). If *x* is less than *y*, the result is 0 (false).

```
if ( !(x < y));
```

Indirection and Address-of Operators

The C indirection and address-of operators are discussed in the following list:

- The indirection operator accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value addressed by the operand; that is, the value at the address specified by the operand. The type of the result is the type that the operand addresses. If the pointer value is invalid, the result is undefined. The specific conditions that invalidate a pointer value are implementation-defined. The following list includes some of the most common:

  - The pointer is a null pointer.
  - The pointer specifies the address of a local item that is not active at the time of the reference.
  - The pointer specifies an address that is inappropriately aligned for the type of the object pointed to.
  - The pointer specifies an address not used by the executing program.

- The address-of operator gives the address of its operand. The operand can be any value that is a valid left-hand value of an assignment operation. A function designator or array name can also be the operand of the address-of operator, although in these cases the operator is superfluous since function designators and array names are addresses. (Assignment operations are discussed in the section entitled "Assignment Operators" later in this chapter.) The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

You cannot apply the address-of operator to a bitfield member of a structure (described in the section entitled "Structure Declarations" in Chapter 4) or to an identifier declared with the **register** storage-class specifier.

Examples 1 through 4 use the following declarations:

```
int *pa, x;
int a[20];
double d;
```

Example 1

In this example, the address-of operator (**&**) takes the address of the sixth element of the array *a*. The result is stored in the pointer variable *pa*.

```
pa = &a[5];
```

Example 2

In this example the indirection operator (*****) is used to access the **int** value at the address stored in *pa*. The value is assigned to the integer variable *x*.

```
x = *pa;
```

Example 3

In this example , the word *True* would be printed. This example demonstrates that the result of applying the indirection operator to the address of *x* is the same as *x*.

```
if (x == *&x)
    printf("True\n");
```

Example 4

This example demonstrates an appropriate application of the rule shown in Example 3. First the address of *x* is converted by a type cast to a pointer to a **double** type; then the indirection operator is applied to give a result of type **double**.

```
d = *(double *)(&x);
```

Example 5

In this example, the function *roundup* is declared, and then two pointers to *roundup* are declared and initialized. The first pointer *proundup* is initialized using only the name of the function, while the second, *pround*, uses the address-of operator in the initialization. The initializations are equal.

```
int roundup() ;

int (*proundup) = roundup;
int (*pround) = &roundup;
```

The sizeof Operator

The **sizeof** operator gives the amount of storage, in bytes, associated with an identifier or a type. This operator lets you avoid specifying machine-dependent data sizes in your programs.

A **sizeof** expression has the form

sizeof *expression*

An *expression* is either an identifier or a type-cast expression (that is, a type specifier enclosed in parentheses). If *expression* is a type-cast expression, it cannot be **void**. If it is an identifier, it cannot represent a bitfield object or a function designator.

When you apply the **sizeof** operator to an array identifier, the result is the size of the entire array rather than the size of the pointer represented by the array identifier.

When you apply the **sizeof** operator to a structure or union type name, or to an identifier of structure or union type, the result is the actual size of the structure or union. This size may include internal and trailing padding used to align the members of the structure or union on memory boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the individual members.

Example 1

This example uses the **sizeof** operator to pass the size of an **int**, which varies among machines, as an argument to a function named *calloc*. The *buffer* stores the value returned by the function.

```
buffer = calloc(100, sizeof (int) );
```

Example 2

In this example, *strings* is an array of pointers to **char**. The number of pointers is the number of elements in the array, but is not specified. It is easy to determine the number of pointers by using the **sizeof** operator to calculate the number of elements in the array. The **const** integer value *string_no* is initialized to this number. Because it is a **const** value, *string_no* cannot be modified.

```
static char *strings[] ={
                        "this is string one",
                        "this is string two",
                        "this is string three",
                       };
const int string_no = (sizeof strings)/(sizeof strings[0]);
```

Multiplicative Operators

The multiplicative operators perform multiplication (*), division (/), and remainder (%) operations. The operands of the remainder operator (%) must be integral. The multiplication (*) and division (/) operators can take integral- or floating-type operands; the types of the operands can be different.

The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

Note

> Since the conversions performed by the multiplicative operators do not provide for overflow or underflow conditions, information may be lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

The C multiplicative operators are described as follows:

*	The multiplication operator causes its two operands to be multiplied.
/	The division operator causes the first operand to be divided by the second. If two integer operands are divided and the result is not an integer, it is truncated according to the following rules: • If both operands are positive or unsigned, the result is truncated toward 0. • If either operand is negative, the direction of truncation of the result (either toward 0 or away from 0) is defined by the implementation. For more information, see your compiler guide. The result of division by 0 is undefined.
%	The result of the remainder operator is the remainder when the first operand is divided by the second. If either or both operands are positive or unsigned, the result is positive. If either operand is negative the sign of the result is defined by the implementation. (For more information, see your compiler guide.) If the right operand is zero, the result is undefined.

These declarations are used for all of the following examples:

```
int i = 10, j = 3, n;
double x = 2.0, y;
```

Example 1

In this example, *x* is multiplied by *i* to give the value 20.0. The result has **double** type.

```
y = x * i;
```

Example 2

In this example, 10 is divided by 3. The result is truncated toward 0, yielding the integer value 3.

```
n = i / j;
```

Example 3

In this example, *n* is assigned the integer remainder, 1, when 10 is divided by 3.

```
n = i % j;
```

Additive Operators

The additive operators perform addition (+) and subtraction (-). The operands can be integral or floating values. Some additive operations can also be performed on pointer values, as outlined under the discussion of each operator.

The additive operators perform the usual arithmetic conversions on integral and floating operands. The type of the result is the type of the operands after conversion. Since the conversions performed by the additive operators do not provide for overflow or underflow conditions, information may be lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

Addition (+)

The addition operator (+) causes its two operands to be added. Both operands can have integral or floating types, or one operand can be a pointer and the other an integer.

When an integer is added to a pointer, the integer value (*i*) is converted by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value representing the address *i* positions from the original address. The new pointer value addresses a value of the same type as the original pointer value.

Subtraction (-)

The subtraction operator (-) subtracts the second operand from the first. The following combinations of operands can be used with this operator:

- Both operands integral or floating type values
- Both operands pointer values to the same type
- The first operand a pointer value and the second operand an integer

When two pointers are subtracted, the difference is converted to a signed integral value by dividing the difference by the size of a value of the type that the pointers address. The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array, as discussed in ''Pointer Arithmetic,'' later in this section.

When an integer value is subtracted from a pointer value, the subtraction operator converts the integer value (*i*) by multiplying it by the size of the value that the pointer addresses. After conversion, the integer value represents *i* memory positions, where each position has the length specified by the pointer type. When the converted integer value is subtracted from the pointer value, the result is the memory address *i* positions before the original address. The new pointer points to a value of the type addressed by the original pointer value.

Pointer Arithmetic

Additive operations involving a pointer and an integer give meaningful results only if the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. When the integer value is converted to an address offset, the compiler assumes that only memory positions of the same size lie between the original address and the address plus the offset.

This assumption is valid for array members. By definition, an array is a series of values of the same type; its elements reside in contiguous memory locations. However, storage for any types except array elements is not guaranteed to be completely filled. That is, blanks may appear between memory positions, even positions of the same type. Therefore, the results of adding to or subtracting from the addresses of any values but array elements are undefined.

Similarly, when two pointer values are subtracted, the conversion assumes that only values of the same type, with no blanks, lie between the addresses given by the operands.

On machines with segmented architecture (such as the 8086/8088), additive operations between pointer and integer values may not be valid in some cases. For example, an operation may result in an address that is outside the bounds of an array. See your compiler guide for more information on memory models.

The following declarations are used for both examples:

```
int i = 4, j;
float x[10];
float *px;
```

Example 1

In this example, the value of *i* is multiplied by the length of a **float** and added to *&x[4]*. The resulting pointer value is the address of *x[8]*.

```
px = &x[4] + i;  /* equivalent to px = &x[4+i]; */
```

Example 2

In this example, the address of the third element of *x* (given by *x[i–2]*) is subtracted from the address of the fifth element of *x* (given by *x[i]*). The difference is divided by the length of a **float**; the result is the integer value 2.

```
j = &x[i] - &x[i-2];
```

Shift Operators

The shift operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies. Both operands must be integral values. These operators perform the usual arithmetic conversions; the type of the result is the type of the left operand after conversion.

For leftward shifts, the vacated right bits are set to 0. For rightward shifts, the vacated left bits are filled based on the type of the first operand after conversion. If the type is **unsigned**, they are set to 0. Otherwise, they are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative.

Since the conversions performed by the shift operators do not provide for overflow or underflow conditions, information may be lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

Example

```
unsigned int x, y, z;

x = 0x00aa;
y = 0x5500;

z = (x << 8) + (y >> 8);
```

In this example, *x* is shifted left eight positions and *y* is shifted right eight positions. The shifted values are added, giving *0xaa55*, and assigned to *z*.

Relational Operators

The binary relational operators compare their first operand to their second operand to test the validity of the specified relationship. The result of a relational expression is 1 if the tested relationship is true and 0 if it is false. The type of the result is **int**.

The relational operators test the following relationships:

<	First operand less than second operand
>	First operand greater than second operand
<=	First operand less than or equal to second operand
>=	First operand greater than or equal to second operand
==	First operand equal to second operand
!=	First operand not equal to second operand

The operands can have integral, floating, or pointer type. The types of the operands can be different. Relational operators perform the usual arithmetic conversions on integral and floating type operands. In addition, you can use the following combinations of operand types with relational operators:

- Both operands of any relational operator can be pointers to the same type. For the equality (==) and inequality (!=) operators, the result of the comparison indicates whether or not the two pointers address the same memory location. For the other relational operators (<, >, <=, and >=), the result of the comparison indicates the relative position of two memory addresses.

  Since the address of a given value is arbitrary, comparisons between the addresses of two unrelated values are generally meaningless. However, comparisons between the addresses of different elements of the same array can be useful, since array elements are guaranteed to be stored in order from the first element to the last. The address of the first array element is "less than" the address of the last element.

- A pointer value can be compared to the constant value 0 for equality (==) or inequality (!=). A pointer with a value of 0, called a "null" pointer, does not point to a memory location.

Example 1

Because *x* and *y* are equal, the expression in Example 1 yields the value 0.

```
int x = 0, y = 0;
x < y
```

Example 2

The fragment in Example 2 initializes each element of *array* to a null character constant.

```
char array[10] ;
char *p ;

  for (p = array; p < &array[10]; p++)
      *p = '\0' ;
```

Example 3

Example 3 declares an enumeration variable named *col* with the tag *color*. At any time, the variable may contain an integer value of 0, 1, or 2, which represents one of the elements of the enumeration set *color*: the color red, white, or green, respectively. If *col* contains 0 when the **if** statement is executed, any statements depending on the **if** will be executed.

```
enum color {red, white, green} col;
         .
         .
         .
         if (col == red)
         .
         .
         .
```

Bitwise Operators

The bitwise operators perform bitwise-AND (**&**), inclusive-OR (/), and exclusive-OR (^) operations. The operands of bitwise operators must have integral types, but their types can be different. These operators perform the usual arithmetic conversions; the type of the result is the type of the operands after conversion.

The C bitwise operators are described as follows:

&	The bitwise-AND operator compares each bit of its first operand to the corresponding bit of its second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
|	The bitwise-inclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If either bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.
^	The bitwise-exclusive-OR operator compares each bit of its first operand to the corresponding bit of its second operand. If one bit is 0 and the other bit is 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

The following declarations are used for these examples:

```
short i = 0xab00;
short j = 0xabcd;
short n;
```

Example 1

The result assigned to *n* in Example 1 is the same as *i* (0xab00 hexadecimal).

```
n = i & j;
```

Example 2

The bitwise-inclusive OR in Example 2 results in the value 0xabcd (hexadecimal).

```
n = i | j;
```

Example 3

The bitwise-exclusive OR in Example 3 produces 0xcd (hexadecimal).

```
n = i ^ j;
```

Logical Operators

The logical operators perform logical-AND (**&&**) and logical-OR (*||*) operations. The operands of the logical operators must have integral, floating, or pointer type. The types of the operands can be different.

The operands of logical-AND and logical-OR expressions are evaluated from left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated. There is a sequence point after the first operand.

Logical operators do not perform the usual arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to 0.

The result of a logical operation is either 0 or 1. The result's type is **int**.

The C logical operators are described as follows:

&&	The logical-AND operator produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation is equal to 0, the second operand is not evaluated.
||	The logical-OR operator performs an inclusive-OR operation on its operands. The result is 0 if both operands have 0 values. If either operand has a nonzero value, the result is 1. If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.

The following examples use these declarations:

```
int w, x, y, z;
```

Example 1

In this example, the *printf* function is called to print a message if *x* is less than *y* and *y* is less than *z*. If *x* is greater than *y*, the second operand ($y < z$) is not evaluated and nothing is printed. Note that this could cause problems in cases where the second operand has side effects that are being relied on for some other reason.

```
if (x < y && y < z)
        printf ("x is less than z\n");
```

Example 2

In this example, if *x* is equal to either *w*, *y*, or *z*, the second argument to the *printf* function evaluates to true and the value 1 is printed. Otherwise, it evaluates to false and the value 0 is printed. As soon as one of the conditions evaluates to true, evaluation ceases.

```
printf ("%d" , (x==w || x==y || x==z));
```

Sequential-Evaluation Operator

The sequential-evaluation operator evaluates its two operands sequentially from left to right. There is a sequence point after the first operand. The result of the operation has the same value and type as the right operand. Each operand can be of any type. The sequential-evaluation operator does not perform type conversions between its operands.

The sequential-evaluation operator, also called the ‘‘comma operator,’’ is typically used to evaluate two or more expressions in contexts where only one expression is allowed.

Commas can be used as separators in some contexts. However, you must be careful not to confuse the use of the comma as a separator with its use as an operator; the two uses are completely different.

Example 1

In this example, each operand of the **for** statement’s third expression is evaluated independently. The left operand, *i += i,* is evaluated first; then the right operand, *j– –*, is evaluated.

```
for ( i = j = 1; i + j < 20; i += i, j--);
```

Example 2

In the function call to *func_one*, three arguments, separated by commas, are passed: *x*, *y* + 2, and *z*.

In the function call to *func_two*, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to *func_two*. The first argument is the result of the sequential-evaluation operation *(x--, y + 2)*, which has the value and type of the expression *y* + 2; the second argument is *z*.

```
func_one(x, y + 2, z);
func_two((x--, y + 2), z);
```

Conditional Operator

C has one ternary operator: the conditional operator (**? :**). It has the following form:

operand1 **?** *operand2* **:** *operand3*

The expression *operand1* must have integral, floating, or pointer type. It is evaluated in terms of its equivalence to 0. A sequence point follows *operand1*. Evaluation proceeds as follows:

- If *operand1* does not evaluate to 0, *operand2* is evaluated, and the result of the expression is the value of *operand2*.
- If *operand1* evaluates to 0, *operand3* is evaluated, and the result of the expression is the value of *operand3*.

Note that either *operand2* or *operand3* is evaluated, but not both.

The type of the result of a conditional operation depends on the type of *operand2* or *operand3*, as follows:

- If *operand2* or *operand3* has integral or floating type (their types can be different), the operator performs the usual arithmetic conversions. The type of the result is the type of the operands after conversion.
- If both *operand2* and *operand3* have the same structure, union, or pointer type, the type of the result is the same structure, union, or pointer type.

- If both operands have type **void**, the result has type **void**.
- If either operand is a pointer to an object of any type, and the other operand is a pointer to **void**, the pointer to the object is converted to a pointer to **void** and the result is a pointer to **void**.
- If either *operand2* or *operand3* is a pointer and the other operand is a constant expression with the value 0, the type of the result is the pointer type.

Example 1

This example assigns the absolute value of *i* to *j*. If *i* is less than 0, -*i* is assigned to *j*. If *i* is greater than or equal to 0, *i* is assigned to *j*.

```
j = (i < 0) ? (-i) : (i);
```

Example 2

In this example, two functions, *f1* and *f2*, and two variables, *x* and *y*, are declared. Later in the program, if the two variables have the same value, the function *f1* is called. Otherwise, *f2* is called.

```
void f1(void) ;
void f2(void) ;
int x ;
int y ;
     .
     .
     .
(x==y) ? (f1()) : (f2()) ;
```

Assignment Operators

The assignment operators in C can both transform and assign values in a single operation. Using a compound-assignment operator to replace two separate operations can make your programs smaller and more efficient.

C provides the following assignment operators:

++	Unary increment
—	Unary decrement
=	Simple assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
+=	Addition assignment
−=	Subtraction assignment
<<=	Left-shift assignment
>>=	Right-shift assignment
&=	Bitwise-AND assignment
|=	Bitwise-inclusive-OR assignment
^=	Bitwise-exclusive-OR assignment

In assignment, the type of the right-hand value is converted to the type of the left-hand value. The specific conversion path, which depends on the two types, is outlined in detail in the section entitled ‘‘Assignment Conversions’’ to be found in the section entitled ‘‘Type Conversions’’ later in this chapter.

Lvalue Expressions

An assignment operation assigns the value of the right-hand operand to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression that refers to a modifiable memory location.

Expressions that refer to memory locations are called ‘‘lvalue expressions.’’ Expressions referring to modifiable locations are ‘‘modifiable lvalues.’’ One example of a modifiable lvalue expression is a variable name declared without the **const** specifier (non-**const**). The name of the variable denotes a storage location, while the value of the variable is the value stored at that location.

The following C expressions may be lvalue expressions:

- An identifier of integral, floating, pointer, structure, or union type
- A subscript ([ ]) expression that does not evaluate to an array or a function
- A member-selection expression (–> or .), if the selected member is one of the aforementioned expressions
- A unary-indirection (*) expression that does not refer to an array or function
- An lvalue expression in parentheses
- A **const** object (a nonmodifiable lvalue)

Note

Microsoft C includes an extension to the ANSI C standard allowing a type cast to a pointer type as an lvalue expression, as long as the size of the object does not change. The following example illustrates this feature:

```
char *p ;
int  i;
long l;

(long *) p = &l ;        /* legal cast */
(long) i = l ;           /* illegal cast */
```

See your compiler guide for information on enabling and disabling the Microsoft extensions.

Unary Increment and Decrement

The unary assignment operators (++ and ––) increment and decrement their operand, respectively. The operand must have integral, floating, or pointer type and must be a modifiable (non-**const**) lvalue expression.

An operand of integral or floating type is incremented or decremented by the integer value 1. The type of the result is the same as the operand type. An operand of pointer type is incremented or decremented by the size of the object it addresses.
An incremented pointer points to the next object; a decremented pointer points to the previous object.

An increment (++) or decrement (––) operator can appear either before or after its operand, with the following results:

- When the operator appears before its operand, the operand is incremented or decremented and its new value is the result of the expression.

- When the operator appears after its operand, the immediate result of the expression is the value of the operand *before* it is incremented or decremented. After that result is applied in context, the operand is incremented or decremented.

Example 1

In this example, the variable *pos* is compared to 0, then incremented. If *pos* was positive before being incremented, the next statement is executed. First, the value of *q* is assigned to *p*. Then *q* and *p* are incremented.

```
if (pos++ > 0)
        *p++ = *q++;
```

Example 2

In this example, the variable *i* is decremented before it is used as a subscript to *line*.

```
if (line[--i] != '\n')
        return;
```

Simple Assignment

The simple-assignment operator assigns its right operand to its left operand. The conversion rules for assignment apply

Example

In this example, the value of *y* is converted to **double** type and assigned to *x*:

```
double x;
int y;

x = y;
```

Compound Assignment

The compound-assignment operators combine the simple-assignment operator with another binary operator. Compound-assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. For example, a compound-assignment expression such as

expression1 += expression2

can be understood as

expression1 = expression1 + expression2

However, the compound-assignment expression is not equivalent to the expanded version because the compound-assignment expression evaluates *expression1* only once, while the expanded version evaluates *expression1* twice: in the addition operation and in the assignment operation.

The operands of a compound-assignment operator must be of integral or floating type. Each compound-assignment operator performs the conversions that the corresponding binary operator performs and restricts the types of its operands accordingly. The addition-assignment (+=) and subtraction-assignment (-=) operators may also have a left operand of pointer type, in which case the right-hand operand must be of integral type. The result of a compound-assignment operation has the value and type of the left operand.

Example

In this example, a bitwise-inclusive-AND operation is performed on *n* and *MASK*, and the result is assigned to *n*. The manifest constant *MASK* is defined with a **#define** preprocessor directive (this directive is discussed in the section entitled ''Manifest Constants and Macros'' in Chapter 8).

```
#define MASK   0xff00

n &= MASK;
```

Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in expressions. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first.

Table 5.1 summarizes the precedence and associativity of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together in a line or large brace, they have equal precedence and are evaluated according to their associativity.

Table 5.1

Precedence and Associativity of C Operators

Symbol[a]	Type of Operation	Associativity
( ) [ ] . –>	Expression	Left to right
– ~ ! * & ++ -- **sizeof** *casts*	Unary[b]	Right to left
* / %	Multiplicative	Left to right
+ –	Additive	Left to right
<< >>	Shift	Left to right
< > <= >=	Relational (inequality)	Left to right
== !=	Relational (equality)	Left to right
&	Bitwise AND	Left to right
^	Bitwise-exclusive OR	Left to right
|	Bitwise-inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Conditional	Right to left

(*Continued on next page*)

Table 5.1

Precedence and Associativity of C Operators (*Continued*)

Symbol[a]	Type of Operation	Associativity
= *= /= %= += -= <<= >>= &= |= ^=	Simple and compound assignment[c]	Right to left
,	Sequential evaluation	Left to right

[a] Operators are listed in descending order of precedence. If several operators appear in the same line or in a large brace, they have equal precedence.

[b] All unary operators have equal precedence.

[c] All simple and compound-assignment operators have equal precedence.

As Table 5.1 shows, operands consisting of a constant, an identifier, a string, a function call, a subscript expression, a member-selection expression, or a parenthetical expression have the highest precedence and associate from left to right. Type-cast conversions have the same precedence and associativity as the unary operators.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either from right to left or from left to right. The direction of evaluation does not affect the results of expressions that include more than one multiplication (*), addition (+), or binary-bitwise (& | ^) operator at the same level. The compiler is free to evaluate such expressions in any order, even when parentheses in the expression appear to specify a particular order. Only the sequential-evaluation (,), logical-AND (&&), logical-OR (||), ternary (?:) and function-call operators constitute sequence points, and therefore guarantee a particular order of evaluation for their operands. The function-call operator is the set of parentheses following the function identifier. The sequential-evaluation operator (,) is guaranteed to evaluate its operands from left to right. (Note that the comma separating arguments in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.) Sequence points are discussed in the section entitled ''Precedence and Order of Evaluation'' later in this chapter.

The unary plus operator (+) is intended to force specific groupings in certain situations. It is implemented syntactically, but not semantically. For further information on unary operators, see the section earlier in this chapter entitled ‘‘Complement and Unary Plus Operators’’.

Logical operators also guarantee evaluation of their operands from left to right. However, they evaluate the smallest number of operands needed to determine the result of the expression. Thus, some operands of the expression may not be evaluated. For example, in the expression *x && y++*, the second operand, *y++*, is evaluated only if *x* is true (nonzero). Thus, *y* is not incremented if *x* is false (0).

The following list shows the default groupings for several sample expressions:

a & b || c	*(a & b) || c*
a = b || c	*a = (b || c)*
q && r || s--	*(q && r) || s--*

In the first expression, the bitwise-AND operator (**&**) has higher precedence than the logical-OR operator (||), so *a & b* forms the first operand of the logical-OR operation.

In the second expression, the logical-OR operator (||) has higher precedence than the simple-assignment operator (=), so *b || c* is grouped as the right-hand operand in the assignment. Note that the value assigned to *a* is either 0 or 1.

The third expression shows a correctly formed expression that may produce an unexpected result. The logical-AND operator (**&&**) has higher precedence than the logical-OR operator (||), so *q && r* is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, *q && r* is evaluated before *s*––. However, if *q && r* evaluates to a nonzero value, *s*–– is not evaluated, and *s* is not decremented. To correct this problem, *s*–– should appear as the first operand of the expression, or *s* should be decremented in a separate operation.

The following expression is illegal and produces a diagnostic message at compile time:

p == 0 ? p += 1: p += 2	*(p == 0 ? p += 1 : p) += 2*

In this expression, the equality operator (==) has the highest precedence, so *p == 0* is grouped as an operand. The ternary operator (**? :**) has the next-highest precedence. Its first operand is *p == 0*, and its second operand is *p += 1*. However, the last operand of the ternary operator is considered to be *p* rather than *p* += *2*, since this occurrence of *p* binds more closely to the ternary operator than it does to the compound-assignment operator. A syntax error occurs because += *2* does not have a left-hand operand. You should use parentheses to prevent errors of this kind and produce more readable code. For example, you could use parentheses as shown to correct and clarify the preceding example:

```
(p == 0) ? (p += 1) : (p += 2)
```

Type Conversions

Type conversions are performed in the following cases:

- When a value of one type is assigned to a variable of a different type
- When a value of one type is explicitly cast to a different type
- When an operator converts the type of its operand or operands before performing an operation
- When a value is passed as an argument to a function

The rules for each kind of conversion are outlined later in this section.

Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable that receives the assignment. C allows conversions by assignment between integral and floating types, even if information is lost in the conversion. The conversion methods used depend on the types involved in the assignment, as described elsewhere in this section as well as in the section named ''Usual Arithmetic Conversion'' to be found in the section entitled ''C Operators'' earlier in this chapter.

Conversions from Signed Integral Types

A signed integer is converted to a shorter signed integer by truncating the high-order bits, and to a longer signed integer by sign extension.

When a signed integer is converted to an unsigned integer, the signed integer is converted to the size of the unsigned integer, and the result is interpreted as an unsigned value.

No information is lost when a signed integer is converted to a floating value, except that some precision may be lost when a **long int** or **unsigned long int** value is converted to a **float** value.

Table 5.2 summarizes conversions from signed integral types. This table assumes that the **char** type is signed by default. If you use a compile-time option to change the default for the **char** type to unsigned, the conversions given in Table 5.3 for the **unsigned char** type apply instead of the conversions in Table 5.2.

Table 5.2

Conversions from Signed Integral Types

From	To	Method
char[a]	**short**	Sign extend
char	**long**	Sign extend
char	**unsigned char**	Preserve pattern; high-order bit loses function as sign bit
char	**unsigned short**	Sign extend to **short**; convert **short** to **unsigned short**
char	**unsigned long**	Sign extend to **long**; convert **long** to **unsigned long**
char	**float**	Sign extend to **long**; convert **long** to **float**
char	**double**	Sign extend to **long**; convert **long** to **double**
char	**long double**	Sign extend to **long**; convert **long** to **double**
short	**char**	Preserve low-order byte
short	**long**	Sign extend
short	**unsigned char**	Preserve low-order byte
short	**unsigned short**	Preserve bit pattern; high-order bit loses function as sign bit
short	**unsigned long**	Sign extend to **long**; convert **long** to **unsigned long**
short	**float**	Sign extend to **long**; convert **long** to **float**

(Continued on next page)

Table 5.2

Conversions from Signed Integral Types (*Continued*)

From	To	Method
short	**double**	Sign extend to **long**; convert **long** to **double**
short	**long double**	Sign extend to **long**; convert **long** to **double**
long	**char**	Preserve low-order byte
long	**short**	Preserve low-order word
long	**unsigned char**	Preserve low-order byte
long	**unsigned short**	Preserve low-order word
long	**unsigned long**	Preserve bit pattern; high-order bit loses function as sign bit
long	**float**	Represent as **float**. If **long** cannot be represented exactly, some precision is lost.
long	**double**	Represent as **double**. If **long** cannot be represented exactly as a **double**, some precision is lost.
long	**long double**	Represent as **double**. If **long** cannot be represented exactly as a **double**, some precision is lost.

[a] All **char** entries assume that the **char** type is signed by default.

Note

The **int** type is equivalent to either the **short** type or the **long** type, depending on the implementation. Conversion of an **int** value proceeds the same as for a **short** or a **long**, whichever is appropriate.

Conversions from Unsigned Integral Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits, or to a longer unsigned or signed integer by zero extending.

When an unsigned integer is converted to a signed integer of the same size, the bit pattern does not change. However, the value it represents changes if the sign bit is set.

Unsigned integer values are converted to floating values by first converting the unsigned integer value to a signed **long** value, then converting that signed **long** value to a floating value.

Table 5.3 summarizes conversions from unsigned integral types.

Table 5.3

Conversions from Unsigned Integral Types

From	To	Method
unsigned char	**char**	Preserve bit pattern; high-order bit becomes sign bit
unsigned char	**short**	Zero extend
unsigned char	**long**	Zero extend
unsigned char	**unsigned short**	Zero extend
unsigned char	**unsigned long**	Zero extend
unsigned char	**float**	Convert to **long**; convert **long** to **float**
unsigned char	**double**	Convert to **long**; convert **long** to **double**
unsigned char	**long double**	Convert to **long**; convert **long** to **double**
unsigned short	**char**	Preserve low-order byte
unsigned short	**short**	Preserve bit pattern; high-order bit becomes sign bit

(*Continued on next page*)

Table 5.3

Conversions from Unsigned Integral Types (*Continued*)

From	To	Method
unsigned short	**long**	Zero extend
unsigned short	**unsigned char**	Preserve low-order byte
unsigned short	**unsigned long**	Zero extend
unsigned short	**float**	Convert to **long**; convert **long** to **float**
unsigned short	**double**	Convert to **long**; convert **long** to **double**
unsigned short	**long double**	Convert to **long**; convert **long** to **double**
unsigned long	**char**	Preserve low-order byte
unsigned long	**short**	Preserve low-order word
unsigned long	**long**	Preserve bit pattern; high-order bit becomes sign bit
unsigned long	**unsigned char**	Preserve low-order byte
unsigned long	**unsigned short**	Preserve low-order word
unsigned long	**float**	Convert to **long**; convert **long** to **float**
unsigned long	**double**	Convert to **long**; convert **long** to **double**
unsigned long	**long double**	Convert to **long**; convert **long** to **double**

Note

> The **unsigned int** type is equivalent either to the **unsigned short** type or to the **unsigned long** type, depending on the implementation. Conversion of an **unsigned int** value proceeds in the same way as conversion of an **unsigned short** or an **unsigned long**, whichever is appropriate.
>
> Conversions from **unsigned long** values to **float**, **double**, or **long double** are not accurate if the value being converted is larger than the maximum positive **long** value.

Conversions from Floating-Point Types

A **float** value converted to a **double** value undergoes no change in value. A **double** value converted to a **float** value is represented exactly, if possible. Precision may be lost if the value cannot be represented exactly.

A floating value is converted to an integral value by first converting to a **long**, then from the **long** value to the specific integral value, as described in Table 5.4. The decimal portion of the floating value is discarded in the conversion to a **long**; if the result is still too large to fit into a **long**, the result of the conversion is undefined.

Table 5.4 summarizes conversions from floating types.

Table 5.4

Conversions from Floating-Point Types

From	To	Method
float	**char**	Convert to **long**; convert **long** to **char**
float	**short**	Convert to **long**; convert **long** to **short**
float	**long**	Truncate at decimal point. If result is too large to be represented as **long**, result is undefined.

(*Continued on next page*)

Table 5.4

Conversions from Floating-Point Types (*Continued*)

From	To	Method
float	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
float	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
float	**double**	Change internal representation
float	**long double**	Change internal representation
double	**char**	Convert to **float**; convert **float** to **char**
double	**short**	Convert to **float**; convert **float** to **short**
double	**long**	Truncate at decimal point. If result is too large to be represented as **long**, result is undefined.
double	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
double	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
double	**float**	Represent as a **float**. If **double** value cannot be represented exactly as **float**, loss of precision occurs. If value is too large to be represented as **float**, the result is undefined.
long double	**char**	Convert to **float**; convert **float** to **char**
long double	**short**	Convert to **float**; convert **float** to **short**
long double	**long**	Truncate at decimal point. If result is too large to be represented as **long**, result is undefined.
long double	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**

(*Continued on next page*)

Table 5.4

Conversions from Floating-Point Types (*Continued*)

From	To	Method
long double	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
long double	**float**	Represent as a **float**. If **double** value cannot be represented exactly as **float**, loss of precision occurs. If value is too large to be represented as **float**, the result is undefined.
long double	**double**	The **long double** value is treated as **double**.

Note

Conversions from **float**, **double**, or **long double** values to **unsigned long** are not accurate if the value being converted is larger than the maximum positive **long** value.

Conversions to and from Pointer Types

A pointer to one type of value can be converted to a pointer to a different type. However, the result may be undefined because of the alignment requirements and sizes of different types in storage.

A pointer to **void** may be converted to or from a pointer to any type, without restriction.

In some implementations, you can use the special keywords **near**, **far**, and **huge** to change the size of pointers within a program. The conversion path depends on your implementation. For example, on an 8086 processor, the compiler might use a segment-register value to convert a 16-bit pointer to a 32-bit pointer. For information about pointer conversions, see your compiler guide.

A pointer value can also be converted to an integral value. The conversion path depends on the size of the pointer and the size of the integral type, according to the following rules:

- If the size of the pointer is greater than or equal to the size of the integral type, the pointer behaves like an unsigned value in the conversion, except that it cannot be converted to a floating value.

- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type. The implementation determines how a pointer is converted to a longer pointer; for information about pointer conversions, see your compiler guide.

Conversely, an integral type can be converted to a pointer type according to the following rules:

- If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer).

- If the size of the integral type is different from the size of the pointer type, the integral type is first converted to the size of the pointer, using the conversion paths given in Tables 5.2 and 5.3. It is then treated as a pointer value.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions may be made on pointer values. In particular, the compiler may make assumptions about the default size of pointers and convert passed pointer values accordingly, unless a forward declaration is present to override the implicit conversion. For information about pointer conversions, see your compiler guide.

Conversions from Other Types

Since an **enum** value is an **int** value by definition, conversions to and from an **enum** value are the same as those for the **int** type. An **int** is equivalent to either a **short** or a **long**, depending on the implementation.

No conversions between structure or union types are allowed.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, and other types cannot be converted to **void** by assignment. However, you can explicitly cast a value to **void** type, as discussed in the next section, ‘‘Type-Cast Conversions’’.

Type-Cast Conversions

You can use type casts to explicitly convert types. A type cast has the form

(*type-name*)*operand*

where *type-name* is a type and *operand* is a value to be converted to that type.

The operand is converted as though it had been assigned to a variable of *type-name* type. The conversion rules for assignments (outlined in the section entitled ‘‘Assignment Conversions’’) apply to type casts as well.

You can use the type name **void** in a cast operation, but you cannot assign the resulting expression to any item.

Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand or operands. Many operators perform the usual arithmetic conversions, outlined in the section entitled ‘‘Usual Arithmetic Conversions’’ earlier in this chapter.

C permits some arithmetic with pointers. In pointer arithmetic, integer values are converted to express memory positions. (For more information, see the discussion of additive operators in the section entitled ‘‘C Operators’’ and the discussion of subscript expressions in the section entitled ‘‘Introduction’’ earlier in this chapter.)

Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on the presence of a function prototype (forward declaration) with declared argument types for the called function.

If a function prototype is present and includes declared argument types, the compiler performs type checking. The type-checking process is outlined in detail in the chapter on ‘‘Functions.’’

If no function prototype is present, or if an old-style forward declaration omits the argument-type list, only the usual arithmetic conversions are performed on the arguments in the function call. These conversions are performed independently on each argument in the call. This means that a

float value is converted to a **double**; a **char** or **short** value is converted to an **int**; and an **unsigned char** or **unsigned short** is converted to an **unsigned int**.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions can also be made on pointer values passed to functions. You can override these implicit conversions by providing function prototypes to let the compiler perform type checking. For information about pointer conversions, see your compiler guide.

Chapter 6

Statements

Introduction 6-1

The break Statement 6-3

The Compound Statement 6-4

The continue Statement 6-5

The do Statement 6-6

The Expression Statement 6-7

The for Statement 6-9

The goto and Labeled Statements 6-11

The if Statement 6-13

The Null Statement 6-15

The return Statement 6-16

The switch Statement 6-18

The while Statement 6-21

Introduction

The statements of a C program control the flow of program execution. In C, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. This chapter describes C statements in alphabetical order, as follows:

- **break** statement
- compound statement
- **continue** statement
- **do** statement
- expression statement
- **for** statement
- **goto** and labeled statements
- **if** statement
- null statement
- **return** statement
- **switch** statement
- **while** statement

C statements consist of keywords, expressions, and other statements. The following keywords appear in C statements:

break
case
continue
default
do
else
for
goto
if
return
switch
while

The expressions in C statements are the expressions discussed in the "Expressions and Assignments" chapter. Statements appearing within C statements may be any of the statements discussed in this chapter. A statement that forms a component of another statement is called the "body" of the enclosing statement. Frequently the statement body is a "compound" statement: a single statement composed of one or more statements.

The compound statement is delimited by braces ({ }); all other C statements end with a semicolon(;).

Any C statement may begin with an identifying label consisting of a name and a colon. Since only the **goto** statement recognizes statement labels, statement labels are described along with the **goto** statement in the section entitled "The goto and Labeled Statements".

Introduction

When a C program is executed, its statements are executed in the order in which they appear in the program, except where a statement explicitly transfers control to another location.

The break Statement

Syntax

break;

Execution

The **break** statement terminates the execution of the smallest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement that follows the terminated statement. A **break** statement can appear only within a **do**, **for**, **switch**, or **while** statement.

Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement that immediately encloses it. You can use a **return** or **goto** statement to transfer control out of the nested structure.

Example

This example processes an array of variable-length strings stored in *lines*. The **break** statement causes an exit from the interior **for** loop after the terminating null character (**'\0'**) of each string is found and its position is stored in *lengths[i]*. Control then returns to the outer **for** loop. The variable *i* is incremented and the process is repeated until *i* is greater than or equal to *LENGTH*.

```
for (i = 0; i < LENGTH; i++) {
     for (j = 0; j < WIDTH; j++) {
          if (lines[i][j] == '\0') {
               lengths[i] = j;
               break;
          }
     }
}
```

The Compound Statement

Syntax

{
[*declaration*]
.
.
.
statement
[*statement*]
.
.
.
}

Execution

A compound statement typically appears as the body of another statement, such as the **if** statement. When a compound statement is executed, its statements are executed in the order in which they appear, except where a statement explicitly transfers control to another location. The ''Declarations'' chapter describes the form and meaning of the declarations that can appear at the head of a compound statement.

Like other C statements, any of the statements in a compound statement can carry a label. Labeled statements are discussed in the section entitled ''The goto and Labeled Statements'' later in this chapter.

Example

In this example, if *i* is greater than 0, all of the statements in the compound statement are executed in order.

```
if (i > 0) {
      line[i] = x;
      x++;
      i--;
}
```

The continue Statement

Syntax

continue;

Execution

The **continue** statement passes control to the next iteration of the **do**, **for**, or **while** statement in which it appears, bypassing any remaining statements in the **do**, **for**, or **while** statement body. The next iteration of a **do**, **for**, or **while** statement is determined as follows:

- Within a **do** or a **while** statement, the next iteration starts by re-evaluating the expression of the **do** or **while** statement.
- Within a **for** statement, the next iteration starts by evaluating the loop expression of the **for** statement. Then it evaluates the conditional expression and, depending on the result, either terminates or iterates the statement body. (The **for** statement is discussed in the section entitled ''The for Statement'' later in this chaper.)

Example

In this example, the statement body is executed if *i* is greater than 0. First *f(i)* is assigned to *x*; then, if *x* is equal to 1, the **continue** statement is executed. The rest of the statements in the body are ignored, and execution resumes at the top of the loop with the evaluation of *i-- > 0*.

```
while (i-- > 0) {
     x = f(i);
     if (x == 1)
          continue;
     y += x * x;
}
```

The do Statement

Syntax

do
statement
while (*expression*);

Execution

The body of a **do** statement is executed one or more times until *expression* becomes false (0). Execution proceeds as follows:

1. The statement body is executed.
2. The *expression* is evaluated. If *expression* is false, the **do** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the process is repeated, beginning with step 1.

The **do** statement may also terminate when a **break**, **goto**, or **return** statement is executed within the statement body.

Example

In this **do** statement, the two statements *y = f(x);* and *x—;* are executed, regardless of the initial value of *x*. Then *x > 0* is evaluated. If *x* is greater than 0, the statement body is executed again and *x > 0* is reevaluated. The statement body is executed repeatedly as long as *x* remains greater than 0. Execution of the **do** statement terminates when *x* becomes 0 or negative. The body of the loop is executed at least once.

```
do {
     y = f(x);
     x--;
} while (x > 0);
```

The Expression Statement

Syntax

expression;

Execution

When an expression statement is executed, the expression is evaluated according to the rules outlined in the ‘‘Expressions and Assignments’’ chapter.

In C, assignments are expressions. The value of the expression is the value being assigned (sometimes called the ‘‘right-hand value’’).

Function calls are also considered expressions. The value of the expression is the value, if any, returned by the function. If a function returns a value, the expression statement usually includes an assignment to store the returned value when the function is called. The value returned by the function is usually used as an operand in another expression. If the value is to be used more than once, it can be assigned to another variable. If the value is neither used as an operand nor assigned, the function is called but the return value, if any, is not used.

Example 1

In this example, *x* is assigned the value of *y + 3*.

```
x = (y + 3);
```

Example 2

In this example, *x* is incremented.

```
x++;
```

Example 3

This example shows a function-call expression. The value of the expression, which includes any value returned by the function, is assigned to the variable *z*.

```
z = f(x) + 3;
```

The for Statement

Syntax

> **for** ([*init-expression*]; [*cond-expression*]; [*loop-expression*])
> *statement*

Execution

The body of a **for** statement is executed zero or more times until the optional *cond-expression* becomes false. You can use the optional *init-expression* and *loop-expression* to initialize and change values during the **for** statement's execution.

Execution of a **for** statement proceeds as follows:

1. The *init-expression*, if any, is evaluated.
2. The *cond-expression*, if any, is evaluated. Three results are possible:
    - If *cond-expression* is true (nonzero), *statement* is executed; then *loop-expression*, if any, is evaluated. The process then begins again with the evaluation of *cond-expression*.
    - If *cond-expression* is omitted, *cond-expression* is considered true, and execution proceeds exactly as described for case *a*. A **for** statement without a *cond-expression* argument terminates only when a **break** or **return** statement within the statement body is executed, or when a **goto** (to a labeled statement outside the **for** statement body) is executed.
    - If *cond-expression* is false, execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement also terminates when a **break, goto,** or **return** statement within the statement body is executed.

Example

This example counts space (**’\x20’**) and tab (**’\t’**) characters in the array of characters named *line* and replaces each tab character with a space. First *i*, *space*, and *tab* are initialized to 0. Then *i* is compared with the constant *MAX*; if *i* is less than *MAX*, the statement body is executed. Depending on the value of *line [i]*, the body of one or neither of the **if** statements is executed. Then *i* is incremented and tested against *MAX*; the statement body is executed repeatedly as long as *i* is less than *MAX*.

```
for (i = space = tab = 0; i < MAX; i++) {
     if (line[i] == ' ')
          space++;
     if (line[i] == '\t') {
          tab++;
          line[i] = ' ';
     }
}
```

The goto and Labeled Statements

Syntax

goto *name*;
.
.
.
name: *statement*

Execution

The **goto** statement transfers control directly to the statement that has *name* as its label. The labeled statement is executed immediately after the **goto** statement is executed. A statement with the given label must reside in the same function, and the given label can appear before only one statement in the same function.

A statement label is meaningful only to a **goto** statement; in any other context, a labeled statement is executed without regard to the label.

A label name is simply an identifier. (The section entitled ‘‘Identifiers’’ in Chapter 2 describes the rules that govern the construction of identifiers.) Each statement label must be distinct from other statement labels in the same function.

Like other C statements, any of the statements in a compound statement can carry a label. Thus, you can use a **goto** statement to transfer into a compound statement. However, transferring into a compound statement is dangerous when the compound statement includes declarations that initialize variables. Since declarations appear before the executable statements in a compound statement, transferring directly to an executable statement within the compound statement bypasses the initializations. The results are undefined.

Example

In this example, a **goto** statement transfers control to the point labeled *exit* if an error occurs.

```
if (errorcode > 0)
    goto exit;
    .
    .
    .
    exit:
    return (errorcode);
```

The if Statement

Syntax

if (*expression*)
    statement1
[**else**
    statement2]

Execution

The body of an **if** statement is executed selectively, depending on the value of *expression*, described as follows:

1. The *expression* is evaluated.
    - If *expression* is true (nonzero), *statement1* is executed.
    - If *expression* is false, *statement2* is executed.
    - If *expression* is false and the **else** clause is omitted, *statement1* is ignored.
2. Control passes from the **if** statement to the next statement in the program.

Example 1

In this example, the statement *y = x/i;* is executed if *i* is greater than 0. If *i* is less than or equal to 0, *i* is assigned to *x* and *f(x)* is assigned to *y*. Note that the statement forming the **if** clause ends with a semicolon.

```
if (i > 0)
     y = x/i;
else {
     x = i;
     y = f(x);
}
```

Note

C does not offer an ‘‘else if’’ statement, but you can achieve the same effect by nesting **if** statements. An **if** statement can be nested within either the **if** clause or the **else** clause of another **if** statement.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. If no braces are present, the compiler resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Example 2

In this example, the **else** clause is associated with the inner **if** statement. If *i* is less than or equal to 0, no value is assigned to *x*.

```
if (i > 0)              /* Without braces */
     if (j > i)
          x = j;
     else
          x = i;
```

Example 3

In this example, the braces surrounding the inner **if** statement make the **else** clause part of the outer **if** statement. If *i* is less than or equal to 0, *i* is assigned to *x*.

```
if (i > 0) {            /* With braces */
     if (j > i)
          x = j;
}
else
     x = i;
```

The Null Statement

Syntax

```
;
```

Execution

A ‘‘null statement’’ is a statement containing only a semicolon; it may appear wherever a statement is expected. Nothing happens when a null statement is executed.

Statements such as **do**, **for**, **if**, and **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body.

As with any other C statement, you can include a label before a null statement. To label an item that is not a statement, such as the closing brace of a compound statement, you can label a null statement and insert it immediately before the item to get the same effect.

Example

In this example, the loop expression of the **for** statement *line[i++]=0* initializes the first 10 elements of *line* to 0. The statement body is a null statement, since no further statements are necessary.

```
for (i = 0; i < 10; line[i++] = 0)
     ;
```

The return Statement

Syntax

return [*expression*];

Execution

The **return** statement terminates the execution of the function in which it appears and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined.

By convention, parentheses enclose the *expression* argument of the **return** statement. However, C does not require the parentheses.

If no **return** statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function is executed. The return value of the called function is undefined. If a return value is not required, declare the function to have **void** return type.

Example

In this example, the *main* function calls two functions: *sq* and *draw*. The *sq* function returns the value of *x* * *x* to *main*, where the return value is assigned to *y*. The *draw* function is declared as a **void** function and does not return a value. An attempt to assign the return value of *draw* would cause a diagnostic message to be issued.

```
main()
{
      void draw(int,int);
      long sq(int);
      .
      .
      .
      y = sq(x);
      draw(x, y);
      .
      .
      .
}

long sq(x)
int x;
{
      return (x * x);
}

void draw(x,y)
int x, y;
{
      .
      .
      .
      return;
}
```

The switch Statement

Syntax

```
switch (expression) {
    [declaration]
    .
    .
    .
    [case constant-expression :]
    .
    .
    .
        [statement]
        .
        .
        .
    [default :
        [statement]]
}
```

Execution

The **switch** statement transfers control to a statement within its body. Control passes to the statement whose **case** *constant-expression* matches the value of **switch** *expression*. The **switch** statement may include any number of **case** instances. Execution of the statement body begins at the selected statement and proceeds until the end of the body or until a statement transfers control out of the body.

The **default** statement is executed if no **case** *constant-expression* is equal to the value of **switch** *expression*. If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body is executed. The **default** statement need not come at the end; it can appear anywhere in the body of the **switch** statement.

The type of **switch** *expression* must be integral, but the resulting value is converted to **int**. Each **case** *constant-expression* is then converted using the usual arithmetic conversions. The value of each **case** *constant-expression* must be unique within the statement body. If the type of **switch** *expression* is larger than **int**, a diagnostic message is issued.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines where execution starts in the statement body. All statements between the statement where execution starts and the end of the body are executed regardless of their labels, unless a statement transfers control out of the body entirely.

Note

> Declarations can appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The **switch** statement transfers control directly to an executable statement within the body, bypassing the lines that contain initializations.

Example 1

In this example, all three statements of the **switch** body are executed if *c* is equal to '*A*'. Execution control is transferred to the first statement (*capa++;*) and continues in order through the rest of the body. If *c* is equal to *lettera and total are incremented. Only total is incremented if c is not equal to or*

```
switch (c) {
        case 'A':
             capa++;
        case 'a':
             lettera++;
        default :
             total++;
      }
```

Example 2

In this example, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the statement body after one statement is executed. If *i* is equal to –1, only *n* is incremented. The **break** following the statement *n++;* causes execution control to pass out of the statement body, bypassing the remaining statements. Similarly, if *i* is equal to 0, only *z* is incremented; if *i* is equal to 1, only *p* is incremented. The final **break** statement is not strictly necessary, since control passes out of the body at the end of the compound statement, but it is included for consistency.

```
switch (i) {
        case -1:
            n++;
            break;
        case 0 :
            z++;
            break;
        case 1 :
            p++;
            break;
      }
```

Multiple Labels

A single statement can carry multiple **case** labels, as the following example shows:

```
case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' :  hexcvt(c);
```

Although you can label any statement within the body of the **switch** statement, no statement is required to carry a label. You can freely intermingle statements with and without labels. Keep in mind, however, that once the **switch** statement passes control to a statement within the body, all following statements in the block are executed, regardless of their labels.

The while Statement

Syntax

while (*expression*)
statement

Execution

The body of a **while** statement is executed zero or more times until *expression* becomes false (0). Execution proceeds as follows:

1. The *expression* is evaluated.

2. If *expression* is initially false, the body of the **while** statement is never executed, and control passes from the **while** statement to the next statement in the program.

   If *expression* is true (nonzero), the body of the statement is executed and the process is repeated beginning at step 1.

The **while** statement may also terminate when a **break**, **goto**, or **return** within the statement body is executed.

Example

This example copies characters from *string2* to *string1*. If *i* is greater than or equal to 0, *string2[i]* is assigned to *string1[i]* and *i* is decremented. When *i* reaches or falls below 0, execution of the **while** statement terminates.

```
while (i >= 0) {
     string1[i] = string2[i];
     i--;
}
```

Chapter 7

Functions

Introduction 7-1

Function Definitions 7-3
    Storage Class 7-4
    Return Type and Function Name 7-5
    Formal Parameters 7-7
    Function Body 7-11

Function Prototypes (Declarations) 7-13

Function Calls 7-16
    Actual Arguments 7-19
    Calls with a Variable Number of Arguments 7-22
    Recursive Calls 7-23

Introduction

A function is an independent collection of declarations and statements, usually designed to perform a specific task. C programs have at least one function, the **main** function, and they may have other functions. This chapter describes how to define, declare, and call C functions.

A function *definition* specifies the name of the function, the types and number of its formal parameters, and the declarations and statements that determine what it does. These declarations and statements are called the ''function body.'' The function definition also gives the function's return type and its storage class. If the return type and storage class are not stated explicitly, they default to **int** and **extern**, respectively.

A function *prototype* (or declaration) establishes the name, return type, and storage class of a function fully defined elsewhere in the program. It can also include declarations giving the types and number of the function's formal parameters. The formal parameter declarations can name the formal parameters, although such names go out of scope at the end of the declaration. The storage class **register** can also be specified for a formal parameter.

Example

This example contrasts the concise and clear prototype declaration and definition formats, and illustrates that the function prototype has the same form as the function definition except that the prototype ends with a semicolon instead of a function body.

The compiler uses the prototype or declaration to compare the types of actual arguments in subsequent calls to the function with the function's formal parameters, even in the absence of an explicit definition of the function. Explicit prototypes and declarations are optional for functions whose return type is **int**. However, to ensure correct behavior, you must declare or define functions with other return types before calling them. (Function prototype declarations are discussed further in ''Function Definitions (Prototypes)'' later in this chapter and in the ''Declarations'' chapter.)

If no prototype or declaration is provided, a default prototype is created from whatever information accompanies the first reference to the function name, whether that reference occurs in a call or a definition. However, such a default prototype may not adequately represent a subsequent definition of, or call to, the function.

A function "call" passes execution control from the calling function to the called function. The actual arguments, if any, are passed by value to the called function. Execution of a **return** statement in the called function returns control and possibly a value to the calling function.

Note

> The use of function prototypes is strongly recommended. Sometimes they provide the only basis on which the compiler can enforce correct argument passing. Prototypes let the compiler either diagnose, or handle correctly, argument mismatches that would otherwise be undetectable until program execution.
>
> The Microsoft C Compiler can generate function prototypes automatically from program source files. These can then be stored in a file that can be included in the compilation of the program. See your compiler guide for more information.

```
/** Prototype-Style Function Declarations and Definitions **/

double new_style(int a, double *x);     /* Function
                                       Prototype      */
double alt_style (int, double *);       /* Alternative
                                       Prototype form */
double old_style ();                /* Obsolete
        .                            * form of function
        .                            * declaration
        .                                             */
double new_style(int a, double *real)   /* Prototype-style */
        {                              /* Function         */
          return (*real + a) ;         /* Definition         */
        }

double alt_style(a , real)          /* Old Form of     */
        double *real ;              /* Function       */
        int a ;                     /* Definition     */
    {
        return (*real + a) ;
    }
```

Function Definitions

Syntax

[*sc-specifier*][*type-specifier*] *declarator* ([*formal-parameter-list*])
function-body

A "function definition" specifies the name, formal parameters, and body of a function. It can also stipulate the function's return type and storage class.

The optional *sc-specifier* gives the function's storage class, which must be either **static** or **extern**.

The optional *type-specifier* and mandatory *declarator* together specify the function's return type and name. The *declarator* is a combination of the identifier that names the function and the parentheses following the function name.

The *formal-parameter-list* is a sequence of formal parameter declarations separated by commas. The following syntax illustrates the form of each formal parameter in a formal parameter list.

[**register**] *type-specifier* [*declarator*]
[,...]

The formal parameter list contains declarations for the function's parameters. If no arguments are to be passed to the function, the list should contain the keyword **void**. The empty parentheses form (( )) can be used, but is obsolete and, if used, conveys no information about whether arguments will be passed. The formal parameter list can be full or partial. The second line of the syntax above shows the "ellipsis notation," a comma followed by three periods (,...). A partial formal parameter list can be terminated by the ellipsis notation to indicate that there may be more arguments passed to the function, but no more information is given about them. Type checking is not performed on such arguments. At least one formal parameter must precede the ellipsis notation and the ellipsis notation must be the last token in the formal parameter list. Without the ellipsis notation, the behavior of a function is undefined if it receives parameters in addition to those declared in the formal parameter list.

When a prototype is available, argument checking and conversion are automatically performed. If no information is given concerning the formal parameters, any undeclared arguments simply undergo the usual arithmetic conversions.

The *type-specifier* can be omitted only if **register** storage class is specified for a value of **int** type.

The *function-body* is a compound statement containing local variable declarations, references to externally declared items, and statements.

Note

> The old forms for function declaration and definition are still supported, but considered obsolete. Use of the prototype form is recommended in new code. The old function-definition form is represented in the following syntax:
>
> [*sc-specifier*][*type-specifier*] *declarator* ([*identifier-list*])
> [*parameter-declarations*]
> *function-body*
>
> The *identifier-list* is an optional list of identifiers that the function will use as the names of formal parameters. The *parameter-declaration* arguments establish the types of the formal parameters.

The section entitled ''Function Definitions'' later in this chapter. describes the parts of a function definition in detail.

Storage Class

The storage-class specifier in a function definition gives the function either **extern** or **static** storage class. If a function definition does not include a storage-class specifier, the storage class defaults to **extern**. You can explicitly give the **extern** storage-class specifier in a function definition, but it is not required.

A function with **static** storage class is visible only in the source file in which it is defined. All other functions, whether they are given **extern** storage class explicitly or implicitly, are visible throughout all the source files that make up the program.

If **static** storage class is desired, it must be declared on the first occurrence of a declaration (if any) of the function, and on the definition of the function.

Note

> A Microsoft extension to the ANSI C standard offers some latitude on functions declared without a storage-class specifier. When the extensions are enabled, a function originally declared without a storage class (or with **extern** storage class) is given **static** storage class if the function definition is in the same source file and explicitly specifies **static** storage class. For information on enabling and disabling extensions, see your compiler guide.

Return Type and Function Name

Syntax

[*sc-specifier*][*type-specifier*] *declarator* ([*formal-parameter-list*])

The return type of a function establishes the size and type of the value returned by the function and corresponds to *type-specifier* in the syntax above. The *type-specifier* can specify any fundamental, structure, or union type. If you do not include *type-specifier*, the return type **int** is assumed.

The *declarator* is the function identifier, which may be modified to a pointer type. The parentheses following the identifier establish the item as a function. Functions cannot return arrays or functions, but they can return pointers to any type, including arrays and functions.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. You need not declare functions with **int** return type before you call them, although prototypes are recommended so that correct argument checking will be enabled. However, functions with other return types must be defined or declared before they are called.

A function's return type is used only when the function returns a value. A function returns a value when a **return** statement containing an expression is executed. The expression is evaluated, converted to the return value type if necessary, and returned to the point at which the function was called. If no **return** statement is executed, or if the **return** statement does not contain an expression, the return value is undefined. If the calling function expects a return value, the behavior of the program is also undefined.

Example 1

In this example, the return type of *add* is **int** by default. The function has **static** storage class, which means that only functions in the same source file can call it. The formal parameters declared in the header include one **int** value, *x*, for which register storage is requested, and a second **int** value, *y*. The second function, *subtract*, is defined in the old form. Its return type is **int** by default. The formal parameters are declared between the header and the opening brace.

```
 /* prototype-style definition: */

static add (register x, int y)
 {
    return (x+y);
 }

 /* old-style definition: */

subtract (x , y)
    int x, y;
 {
  return (x-y);
 }
```

Example 2

This example defines the *STUDENT* type with a **typedef** declaration and defines the function *sortstu* to have *STUDENT* return type. The function selects and returns one of its two structure arguments. This prototype-style definition has the formal parameters declared in the header. In subsequent calls to the function, the compiler checks to make sure the argument types are *STUDENT*. Efficiency would be enhanced by passing pointers to the structure, rather than the entire structure.

```
typedef struct  {
    char name[20];
    int id;
    long class;
} STUDENT;

      /* return type is STUDENT: */

STUDENT sortstu (STUDENT a, STUDENT b)
{
    return ( (a.id < b.id) ? a : b );
}
```

Example 3

This example uses the old form to define a function returning a pointer to an array of characters. The function takes two character arrays (strings) as arguments and returns a pointer to the shorter of the two strings. A pointer to an array points to the type of the array elements; thus, the return type of the function is pointer to **char**.

```
  /* return type is char pointer: */

char *smallstr(s1, s2)
char s1[], s2[];
{
    int i;

    i=0;
    while ( s1[i] != '\0' && s2[i] != '\0' )
                    i++;
    if ( s1[i] == '\0' )
            return (s1);
    else
            return (s2);
}
```

Formal Parameters

"Formal parameters" are variables that receive values passed to a function by a function call. In a function prototype-style definition, the parentheses following the function name contain complete declarations of the function's formal parameters.

Note

> In the old form of a function definition, the formal parameters were declared following the closing parenthesis, immediately before the beginning of the compound statement constituting the function body. In that form, an identifier list within the parentheses specifies the name of each of the formal parameters and the order in which they take on values in the function call. The identifier list consists of zero or more identifiers, separated by commas. The list must be enclosed in parentheses, even if it is empty. This form is obsolete and should not be used in new code.

If at least one formal parameter occurs in the formal parameter list, the list can end with a comma followed by three periods (,...). This construction, called the ''ellipsis notation,'' indicates a variable number of arguments to the function. However, a call to the function is expected to have at least as many arguments as there are formal parameters before the last comma. In the obsolete definition form, the ellipsis notation can follow the last identifier in the identifier list.

If no arguments are to be passed to the function, the list of formal parameters is replaced by the keyword **void**. This use of **void** is distinct from its use as a type specifier.

Note

> To maintain compatibility with previous versions, a Microsoft extension to the ANSI C standard allows a comma without trailing periods (,) at the end of the list of formal parameters to indicate a variable number of arguments. However, it is recommended that code be changed to incorporate the ellipsis notation. For information on enabling and disabling extensions, see your compiler guide.

Formal parameter declarations specify the types, sizes, and identifiers of values stored in the formal parameters. In the obsolete function definition form, these declarations have the same form as other variable declarations (see the ''Declarations'' chapter). However, in a function prototype-style definition, each identifier in the *formal-parameter-list* must be preceded by its appropriate type specifier. For example, in the following (obsolete form) definition of the function *old*, *double x, y, z ;* can be declared simply by separating identifiers with commas:

```
void old(x, y, z)
     double z, y ;
     double x ;
       {
         ;
       }

void new(double x, double y, double z)
       {
         ;
       }
```

The function called *new* is defined in prototype format, with a list of formal parameters in the parentheses. In this form, the type specifier *double* must be repeated for each identifier.

The order and type of formal parameters, including any use of the ellipsis notation, must be the same in all the function declarations (if any) and in the function definition. The types of the actual arguments in calls to a function must be assignment compatible with the types of the corresponding formal parameters, up to the point of the ellipsis notation. Arguments following the ellipsis are not checked. A formal parameter can have any fundamental, structure, union, pointer, or array type.

The only storage class you can specify for a formal parameter is **register**. Undeclared identifiers in the parentheses following the function name are assumed to have **int** type. In the old function-definition form, formal parameter declarations can be in any order.

The identifiers of the formal parameters are used in the function body to refer to the values passed to the function. These identifiers cannot be redefined in the outermost block of the function body, but they can be redefined in inner, nested blocks.

In the obsolete form, only identifiers appearing in the identifier list can be declared as formal parameters. Functions having variable-length argument lists should use the new prototype form. You are responsible for determining the number of arguments passed, and for retrieving additional arguments from the stack within the body of the function. (For information about macros that let you do this in a portable way, see your compiler guide.)

The compiler performs the usual arithmetic conversions independently on each formal parameter and on each actual argument, if necessary. After conversion, no formal parameter is shorter than an **int**, and no formal parameter has **float** type. This means, for example, that declaring a formal parameter as a **char** has the same effect as declaring it as an **int**.

If the **near**, **far**, and **huge** keywords are implemented, the compiler can also convert pointer arguments to the function. The conversions performed depend on the default size of pointers in the program and the presence or absence of a list of argument types for the function. For specific information about pointer conversions, see your compiler guide.

The converted type of each formal parameter determines the interpretation of the arguments that the function call places on the stack. A type mismatch between an actual argument and a formal parameter can cause the arguments on the stack to be misinterpreted. For example, if a 16-bit pointer is passed as an actual argument, then declared as a **long** formal parameter, the first 32 bits on the stack are interpreted as a **long** formal parameter. This error creates problems not only with the **long** formal parameter, but with any formal parameters that follow it. You can detect errors of this kind by declaring function prototypes for all functions.

Example

This example contains a structure-type declaration, a prototype of the function *match*, a call to *match*, and a prototype-style definition of *match*. Note that the same name, *student*, can be used without conflict both for the structure tag and for the structure variable name.

The *match* function is declared to have two arguments: the first, represented by *r*, is a pointer to the *struct student* type; the second, represented by *n*, is a pointer to a value of type **char**.

In the definition, the two formal parameters of the *match* function are declared in the formal parameter list in the parentheses following the function name, with the identifiers *r* and *n*. The parameter *r* is declared as a pointer to the *struct student* type; the parameter *n* is declared as a pointer to a **char** type value.

The function is called with two arguments, both members of the *student* structure. Because there is a prototype of *match*, the compiler performs type checking between the actual arguments and the types specified in the prototype and between the actual arguments and the formal parameters in the definition. Since the types match, no warnings or conversions are necessary.

Note that the array name given as the second argument in the call evaluates to a **char** pointer. The corresponding formal parameter is also declared as a **char** pointer and is used in subscripted expressions as though it were an array identifier. Since an array identifier evaluates to a pointer expression, the effect of declaring the formal parameter as *char *n* is the same as declaring it *char n[]*.

Within the function, the local variable *i* is defined and used to monitor the current position in the array. The function returns the *id* structure member if the *name* member matches the array *n*; otherwise, it returns 0.

```
struct student {
        char name[20];
        int id;
        long class;
        struct student *nextstu;
} student;

main()
{
      /* declaration of function prototype: */

        int match ( struct student *r, char *n );
        .
        .
        .
        if (match (student.nextstu, student.name) > 0) {
        .
        .
        .
        }
}

/* prototype style function definition */

match ( struct student *r, char *n )
{
        int i = 0;

        while ( r->name[i] == n[i] )
                if ( r->name[i++] == '\0' )
                        return (r->id);
        return (0);
}
```

Function Body

A "function body" is a compound statement containing the statements that define what the function does. It can also contain declarations of variables used by these statements. (The section entitled "The Compound Statement" in Chapter 6 discusses compound statements.)

All variables declared in a function body have **auto** storage class unless otherwise specified. When the function is called, storage is created for the local variables and local initializations are performed. Execution control passes to the first statement in the compound statement and continues sequentially until a **return** statement is executed or the end of the function body is encountered. Control then returns to the point at which the function was called.

A **return** statement containing an expression must be executed if the function is to return a value. The return value of a function is undefined if no **return** statement is executed or if the **return** statement does not include an expression.

Function Prototypes (Declarations)

A "function prototype" declaration specifies the name, return type, and storage class of a function. It can also establish types and identifiers of some or all of the function's arguments. The prototype has the same format as the function definition, except that it is terminated by a semicolon immediately following the closing parenthesis and therefore has no body. (See the "Declarations" chapter for a detailed description of the syntax of function declarations.)

You can declare a function implicitly, or you can use a "function prototype" (sometimes called a "forward declaration") to declare it explicitly. A prototype is a declaration that precedes the function definition. In either case, the return type must agree with the return type specified in the function definition.

If a call to a function precedes its declaration or definition, a default prototype of the function is constructed, giving it **int** return type. The types and number of the actual arguments are used as the basis for declaring the formal parameters. Thus a call to the function is an implicit declaration, but the prototype generated may not adequately represent a subsequent definition of, or call to, the function.

A prototype establishes the attributes of a function so that calls to the function that precede its definition (or occur in other source files) can be checked for argument- and return-type mismatches. If you specify the **static** storage-class specifier in a prototype, you must also specify the **static** storage class in the function definition.

If you specify the **extern** storage-class specifier or omit the storage-class specifier entirely, the function has **extern** class. (For an explanation of the Microsoft extension that offers some latitude in function storage-class specification, see the *Note* in the section entitled "Storage Classes") in Chapter 4.

Function prototypes have the following important uses:

- They establish the return type for functions that return any type other than **int**. If you call such a function before you declare or define it, the results are undefined. Although functions that return **int** values do not require prototypes, they are recommended.

- If the prototype contains a full list of parameter types, the types of the arguments occurring in a function call or definition can be checked. The prototype can include both the type of, and an identifier for, each expression that will be passed as an actual argument. However, such identifiers have scope only until the end of the declaration. The prototype can also reflect the fact that the number of arguments will be variable, or that there will be no arguments passed.

  The parameter list in a prototype is a list of type names, separated by commas, corresponding to the actual arguments in the function call. The list is used for checking the correspondence of actual arguments in the function call with the formal parameters in the function definition. Without such a list, mismatches may not be revealed, so the compiler cannot generate diagnostic messages concerning them. (Type checking is further discussed in the subsection ‘‘Actual Arguments’’ of the section entitled ‘‘Function Calls’’ later in this chapter.)

- Prototypes are used to initialize pointers to functions before those functions are defined.

Example

In this example, the function *intadd* is implicitly declared to return an **int** value, since it is called before it is defined. The compiler creates a prototype using the information in the first call. Therefore, when the second call to *intadd* is encountered, the compiler sees the mismatch between *val1*, which is a **float**, and the **int** type of the first argument in its self-created prototype. The **float** is converted to an **int** and passed. Note that if the calls to *intadd* were reversed, the prototype created would expect a **float** as the first argument to *intadd*. When the second call is made, the variable *a* would be converted at the call, but when the value is actually passed to *intadd*, a diagnostic message would be issued because the **int** type specified in the definition does not match the **float** type in the compiler-created prototype.

The function *realadd* returns a **double** value instead of an **int** value. Therefore, the prototype of *realadd* in the *main* function is necessary because the *realadd* function is called before it is defined. Note that the definition of *realadd* matches the forward declaration by specifying the **double** return type.

The forward declaration of *realadd* also establishes the types of its two arguments. The actual argument types match the types given in the declaration and also match the types of the formal parameters in the definition.

```
main()
{
        int a = 0, b = 1;
        float val1= 2.0, val2 = 3.0;

       /* function prototype: */

        double realadd(double x, double y);

        a = intadd (a, b);     /* first call to intadd */
        val1 = realadd(val1, val2);
       a = intadd(val1,b);     /* second call to intadd */
}

/* functions defined with formal parameters in header: */

intadd(int a, int b)
{
        return (a + b);
}

double realadd(double x, double y)
{
        return (x + y);
}
```

Function Calls

Syntax

expression([*expression-list*])

A ''function call'' is an expression that passes control and actual arguments (if any) to a function. In a function call, *expression* evaluates to a function address and *expression-list* is a list of expressions (separated by commas). The values of these latter expressions are the actual arguments passed to the function. If the function takes no arguments, *expression-list* can be empty.

When the function call is executed:

1. The expressions in *expression-list* are evaluated and converted using the usual arithmetic conversions. If a function prototype is available, the results of these conversions may be further converted consistent with the formal parameter declarations.
2. The expressions in *expression-list* are passed to the formal parameters of the called function. The first expression in the list always corresponds to the first formal parameter of the function, the second expression corresponds to the second formal parameter, and so on through the list. Since the called function uses copies of the actual arguments, any changes it makes to the arguments do not affect the values of variables from which the copies may have been made.
3. Execution control passes to the first statement in the function.
4. The execution of a **return** statement in the body of the function returns control and possibly a value to the calling function. If no **return** statement is executed, control returns to the caller after the last statement of the called function is executed. In such cases, the return value is undefined.

Note

> The expressions in the function argument list can be evaluated in any order, so arguments whose values may be changed by side effects from another argument have undefined values. The sequence point defined by the function-call operator guarantees only that all side effects in the argument list are evaluated before control passes to the called function. See the ‘‘Expressions and Assignments’’ chapter for more information on sequence points.

The only requirement in a function call is that the expression before the parentheses must evaluate to a function address. This means that a function can be called through any function-pointer expression.

A function is called in much the same way it is declared. For instance, when you declare a function, you specify the name of the function, followed by a list of formal parameters in parentheses. Similarly, when a function is called, you need only specify the name of the function, followed by an argument list in parentheses. The indirection operator (*) is not required to call the function because the name of the function evaluates to the function address.

The same principle applies when you call a function using a pointer. For example, suppose a function pointer has the following prototype:

```
int (*fpointer)(int num1, int num2);
```

The identifier *fpointer* is declared to point to a function taking two **int** arguments, represented by *num1 and num2*, respectively, and returning an **int** value. A function call using *fpointer* might look like this:

```
(*fpointer)(3,4)
```

The indirection operator (*) is used to obtain the address of the function to which *fpointer* points. The function address is then used to call the function. If a prototype of the pointer to the function precedes the call, the same checking will be performed as with any other function.

Function Calls

Example 1

In this example, the *realcomp* function is called in the statement *rp = realcomp(a, b);*. Two **double** arguments are passed to the function. The return value, a pointer to a **double** value, is assigned to *rp*.

```
double *realcomp(double value1, double value2);
double a, b, *rp;
.
.
.
rp = realcomp(a, b);
```

Example 2

In this example, the function call in *main* passes an integer variable and the address of the function *lift* to the function *work*:

```
work (count, lift);
```

Note that the function address is passed simply by giving the function identifier, since a function identifier evaluates to a pointer expression. To use a function identifier in this way, the function must be declared or defined before the identifier is used; otherwise, the identifier is not recognized. In this case, a prototype for *work* is given at the beginning of the *main* function.

The formal parameter *function* in *work* is declared to be a pointer to a function taking one **int** argument and returning a **long** value. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a **long** value.

The function *work* calls the selected function by using the following function call:

```
(*function)(i);
```

One argument, *i*, is passed to the called function.

```
main ()
{
    /* function prototypes: */

    long lift(int), step(int), drop(int);
    void work (int number, long (*function)(int i));

    int select, count;
    .
    .
    .
    select = 1;
    switch ( select ) {
           case 1: work(count, lift);
                   break;

           case 2: work(count, step);
                   break;

           case 3: work(count, drop);

           default:
                   break;
     }
}

/* function definition with formal parameters in header: */

void work ( int number, long (*function)(int i) )
{
    int i;
    long j;

    for (i = j = 0; i < number; i++)
            j += (*function)(i);
}
```

Actual Arguments

An actual argument can be any value with fundamental, structure, union, or pointer type. Although you cannot pass arrays or functions as parameters, you can pass pointers to these items.

All actual arguments are passed by value. A copy of the actual argument is assigned to the corresponding formal parameter. The function uses this copy without affecting the variable from which it was originally derived.

Pointers provide a way for a function to access a value by reference. Since a pointer to a variable holds the address of the variable, the function can use this address to access the value of the variable. Pointer arguments allow a function to access arrays and functions, even though arrays and functions cannot be passed as arguments.

The expressions in a function call are evaluated and converted as follows:

- The usual arithmetic conversions are performed on each actual argument in the function call. If a prototype is available, the resulting argument type is compared to the prototype's corresponding formal parameter. If they do not match, either a conversion is performed, or a diagnostic message is issued. The formal parameters also undergo the usual arithmetic conversions.

- If no prototype is available, the usual arithmetic conversions are performed on each actual argument before it is passed to the function. A prototype is created whose formal parameter types correspond to the types of the actual arguments after conversion.

If the **near**, **far**, and **huge** keywords are implemented, implementation-dependent conversions on pointer arguments can also be performed. For information about pointer conversions, see your compiler guide.

The number of expressions in the expression list must match the number of formal parameters, unless the function's prototype or definition explicitly specifies a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the list of formal parameters and converts them, if necessary, as described above.

If the prototype's formal parameter list contains only the keyword **void**, the compiler expects zero actual arguments in the function call and zero formal parameters in the definition. A diagnostic message is issued if it finds otherwise.

The type of each formal parameter also undergoes the usual arithmetic conversions. The converted type of each formal parameter determines how the arguments on the stack are interpreted; if the type of the formal parameter does not match the type of the actual argument, the data on the stack may be misinterpreted.

Note

> Type mismatches between actual arguments and formal parameters can produce serious errors, particularly when the sizes are different. The compiler may not be able to detect these errors unless you declare complete prototypes of functions prior to calling them. In the absence of explicit prototypes, the compiler constructs prototypes from whatever information is available in the first reference to the function.
>
> As an example of a serious error, consider a call to a function with an **int** argument. If the function is defined to take a **long**, and the definition occurs in a different module, the compiler-generated prototype will not match the definition, but the error will not be detected because the separate modules will compile without diagnostic messages.

Example

In this example, the *swap* function is declared in *main* to have two arguments, represented respectively by identifiers *num1 and num2*, both of which are pointers to **int** values. The formal parameters *num1* and *num2* in the prototype-style definition are also declared as pointers to **int** type values. In the following function call the address of *x* is stored in *num1* and the address of *y* is stored in *num2*.

```
swap (&x, &y)
```

Now two names, or ‘‘aliases,’’ exist for the same location. References to **num1* and **num2* in *swap* are effectively references to *x* and *y* in *main*. The assignments within *swap* actually exchange the contents of *x* and *y*. Therefore, no **return** statement is necessary.

The compiler performs type checking on the arguments to *swap* because the prototype of *swap* includes argument types for each formal parameter. The identifiers within the parentheses of the prototype and definition can be the same or different. What is important is that the types of the actual arguments match those of the formal parameter lists in both the prototype and the eventual definition.

```
main ()
{
      /* function prototype: */

        void swap (int *num1, int *num2);
        int x, y;
        .
        .
        .
        swap(&x, &y);
}

/* function definition: */

void swap (int *num1, int *num2)
{
        int t;

        t = *num1;
        *num1 = *num2;
        *num2 = t;
}
```

Calls with a Variable Number of Arguments

To call a function with a variable number of arguments, simply specify any number of arguments in the function call. If there is a prototype declaration of the function, a variable number of arguments can be specified by placing a comma followed by three periods (,...), the ‘‘ellipsis notation,’’ at the end of the formal parameter list or list of argument types (see the section entitled ‘‘Function Prototypes (Declarations)’’). The function call must include one argument for each type name declared in the formal parameter list or the list of argument type.

Similarly, the formal parameter list (or identifier list, in the obsolete form) in the function definition can end with the ellipsis notation to indicate a variable number of arguments. For more information about the form of the formal parameter list, see the section entitled ‘‘Function Definitions’’ earlier in this chapter.

Note

To maintain compatibility with previous versions, a Microsoft extension to the ANSI C standard allows a comma without trailing periods (,) at the end of the list of formal parameters to indicate a variable number of arguments. For information on enabling and disabling extensions, see your compiler guide.

All the arguments specified in the function call are placed on the stack. The number of formal parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the formal parameters. You are responsible for retrieving any additional arguments from the stack and for determining how many arguments are present. For information about macros that you can use to handle a variable number of arguments in a portable way, see your compiler guide.

Recursive Calls

Any function in a C program can be called recursively; that is, it can call itself. The C compiler allows any number of recursive calls to a function. Each time the function is called, new storage is allocated for the formal parameters and for the **auto** and **register** variables so that their values in previous, unfinished calls are not overwritten. Parameters are only directly accessible to the instance of the function in which they are created. Previous parameters are not directly accessible to ensuing instances of the function.

Note that variables declared with **static** storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Although the C compiler does not limit the number of times a function can be called recursively, the operating environment may impose a practical limit. Since each recursive call requires additional stack memory, too many recursive calls can cause a stack overflow.

Chapter 8

Preprocessor Directives and Pragmas

Introduction 8-1

Manifest Constants and Macros 8-3
    Preprocessor Operators 8-3
    The #define Directive 8-4
    The #undef Directive 8-11

Include Files 8-12

Conditional Compilation 8-14
    The #if, #elif, #else, and #endif Directives 8-14
    The #ifdef and #ifndef Directives 8-18

Line Control 8-19

Pragmas 8-21

Introduction

A ‘‘preprocessor directive’’ is an instruction to the C preprocessor. The C preprocessor is a text processor that manipulates the text of a source file as the first phase of compilation. Though the compiler ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked separately to process text without compiling.

Preprocessor directives are typically used to make source programs easy to change and easy to compile in different execution environments. Directives in the source file tell the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, or suppress compilation of part of the file by removing sections of text.

The C preprocessor recognizes the following directives:

#define
#elif
#else
#endif
#if
#ifdef
#ifndef
#include
#line
#undef

The number sign (#) must be the first non-white-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives include arguments or values. Any text that follows a directive (except an argument or value that is part of the directive) must be enclosed in comment delimiters (/* */).

Preprocessor directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear.

A ‘‘preprocessor operator’’ is an operator that is only recognized as an operator within the context of preprocessor directives. There are only three preprocessor-specific operators: the ‘‘stringizing’’ operator (#), the ‘‘token-pasting’’ (##) operator, and the **defined** operator. The first two are discussed in the context of the **#define** directive, later in this chapter. The **defined** operator is also discussed later in this chapter. ‘‘The #if, #elif, #else, and #endif Directives.’’

Introduction

A ‘‘pragma’’ is a ‘‘pragmatic,’’ or practical, instruction to the C compiler. Pragmas in C source files are typically used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. (The ‘‘Pragmas’’ section later in this chapter describes the syntax for pragmas). However, the compiler implementation defines the particular pragmas that are available and their meanings. For information about the use and effects of specific pragmas, see your compiler guide.

Manifest Constants and Macros

The **#define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are called ‘‘manifest constants.’’ Identifiers that represent statements or expressions are called ‘‘macros.’’

Once you have defined an identifier, you cannot redefine it to a different value without first removing the original definition. However, you can redefine the identifier with exactly the same definition. Thus, the same definition can appear more than once in a program.

The **#undef** directive removes the definition of an identifier. Once you have removed the definition, you can redefine the identifier to a different value. The section later in this chapter entitled ‘‘Manifest Constants and Macros’’ discusses the **#define** and **#undef** directives.

In practical terms there are two types of macros. ‘‘Object-like’’ macros take no arguments, while ‘‘function-like’’ macros can be defined to accept arguments so that they look and act like function calls. Because macros do not generate actual function calls, you can make programs faster by replacing function calls with macros. However, macros can create problems if you do not define and use them with care. You may have to use parentheses in macro definitions with arguments to preserve the proper precedence in an expression. Also, macros may not handle expressions with side effects correctly. For more information, see the examples in the section entitled ‘‘The #define Directive.’’

Preprocessor Operators

There are three preprocessor-specific operators, one of which is represented by the number sign (#), one by a double number sign (##), and the third by the word **defined**. The ‘‘stringizing’’ operator (#) preceding a macro formal-parameter name in the body of a preprocessor macro causes the corresponding actual argument to be enclosed in string quotation marks. The ‘‘token-pasting’’ operator (##) allows tokens used as actual arguments to be concatenated to form other tokens. These two operators are used in the context of the **#define** directive and are described under ‘‘The #define Directive’’ in the section entitled ‘‘Manifest Constants and Macros’’.

Finally, the **defined** operator simplifies the writing of compound expressions in certain macro directives. It is used in conditional compilation, and is therefore discussed in the section entitled ''Conditional Compilation''. ''The #if, #elif, #else, and #endif Directives.''

The #define Directive

Syntax

#define *identifier substitution-text*
#define *identifier*(*parameter-list*) *substitution-text*

The **#define** directive substitutes *substitution-text* for all subsequent occurrences of *identifier* in the source file. The *identifier* is replaced only when it forms a token. (Tokens are described in the ''Elements of C'' chapter and in the ''Syntax Summary.'') For instance, *identifier* is not replaced if it appears within a string or as part of a longer identifier.

If *parameter-list* appears after *identifier*, the **#define** directive replaces each occurrence of *identifier*(*parameter-list*) with a version of the *substitution-text* argument that has actual arguments substituted for formal parameters.

The *substitution-text* argument consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate *substitution-text* from *identifier* (or from the closing parenthesis following *parameter-list*). This white space is not considered part of the substituted text, nor is any white space following the last token of the text. Text longer than one line can be continued onto the next line by placing a backslash (\) before the new-line character.

The *substitution-text* argument can also be empty. Choosing this option removes occurrences of *identifier* from the source file. The *identifier* is still considered defined, however, and yields the value 1 when tested with the **#if** directive (discussed in the section entitled ''The #if, #elif, #else, and #endif Directives'').

The optional *parameter-list* consists of one or more formal parameter names separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces can separate *identifier* and the opening parenthesis. The scope of a formal parameter name extends to the new line that ends *substitution-text*.

Formal parameter names appear in *substitution-text* to mark the places where actual values will be substituted. Each parameter name can appear more than once in *substitution-text*, and the names can appear in any order.

The actual arguments following an instance of *identifier* in the source file are matched to the corresponding formal parameters of *parameter-list*. Each formal parameter in *substitution-text* that is not preceded by a stringizing (#) or token-pasting (##) operator, or followed by a ## operator, is replaced by the corresponding actual argument. Any macros in the actual argument will be expanded before it replaces the formal parameter. (The # and ## operators are described in the section entitled ''The #define Directive'' later in this chapter.) The actual-argument list must have the same number of arguments as *parameter-list*.

If the name of the macro being defined occurs in *substitution-text* (even as a result of another macro expansion), it is not expanded.

Arguments with side effects sometimes cause macros to produce unexpected results. A given formal parameter may appear more than once in *substitution-text*. If that formal parameter is replaced by an expression with side effects, the expression, with its side effects, may be evaluated more than once.

Stringizing Operator (#)

The number-sign or ''stringizing'' operator (#) is used only with macros that take arguments. If it precedes a formal parameter in the macro definition, the actual argument passed by the macro invocation is enclosed in quotation marks and treated as a string literal. The string literal then replaces each occurrence of a combination of the stringizing operator and formal parameter within the macro definition. White space preceding the first token of the actual argument and following the last token of the actual argument is ignored. Any white space between the tokens in the actual argument is reduced to a single white space in the resulting string literal. Thus, if a comment occurs between two tokens in the actual argument, it is reduced to a single white space. The resulting string literal is automatically concatenated with any adjacent string literals from which it is separated only by white space. Furthermore, if a character contained in the argument normally requires an escape sequence when used in a string literal—for example, the quotation-mark (") or backslash (\) characters—the necessary escape backslash is automatically inserted before the character. The following example shows a macro definition that includes the stringizing operator and a main function that invokes the macro:

```
#define stringer(x) printf(#x "\n")

main()
{
  stringer (I will be in quotes in the printf function call);
  stringer ("I will be in quotes when printed to the screen");
  stringer (This: \"  prints an escaped double quote mark);
}
```

Such invocations would be expanded during preprocessing, producing the following code:

```
printf("I will be in quotes in the printf function call" "\n");
printf("\"I will be in quotes when printed to the screen\"" "\n");
printf("This \\\" prints an escaped double quote mark");
```

When the program is run, screen output for each line would be as follows:

```
I will be in quotes in the printf function call

"I will be in quotes when printed to the screen"

This: \" prints an escaped double quote mark
```

Note

> The Microsoft extension to the ANSI C standard that previously enabled expansion of macro formal arguments appearing in string literals and character constants is no longer supported. Code that relied on this extension should be rewritten using the stringizing (#) operator.

Token-Pasting Operator (##)

The double-number-sign or ‘‘token-pasting’’ operator (##) is used in both object-like and function-like macros. It permits separate tokens to be joined into a single token, and therefore cannot be the first or last token in the macro definition.

If a formal parameter in a macro definition is preceded or followed by the token-pasting operator, the formal parameter is immediately replaced by the unexpanded actual argument. Macro expansion is not performed on the argument prior to replacement. Then, each occurrence of the token-pasting operator in *substitution-text* is removed, and the tokens preceding and following it are concatenated. The resulting token must be a valid token. If it is, the token is rescanned for possible replacement if it represents a macro name. Example 7 shows how tokens can be pasted together using the token-pasting operator.

Example 1

This example defines the identifier *WIDTH* as the integer constant 80 and defines *LENGTH* in terms of *WIDTH* and the integer constant 10. Each occurrence of *LENGTH* is replaced by (*WIDTH + 10*). In turn, each occurrence of *WIDTH + 10* is replaced by the expression (*80 + 10*).

```
#define WIDTH       80
#define LENGTH      (WIDTH + 10)
```

The parentheses around *WIDTH + 10* are important because they control the interpretation in statements such as the following:

```
var = LENGTH * 20;
```

After the preprocessing stage the statement becomes

```
var = (80 + 10) * 20;
```

which evaluates to 1800. Without parentheses, the result is

```
var = 80 + 10 * 20;
```

which evaluates to 280.

Example 2

This example defines the identifier The definition is extended *FILEMESSAGE.* to a second line by using the convention of a backslash followed by a new-line character.

```
#define FILEMESSAGE "Attempt to create file \
failed because of insufficient space"
```

Example 3

This example defines three identifiers, *REG1*, *REG2*, and *REG3*. *REG1* and *REG2* are defined as the keyword **register**. The definition of *REG3* is empty, so each occurrence of *REG3* is removed from the source file. These directives can be used to ensure that the program's most important variables (declared with *REG1* and *REG2*) are given **register** storage. (For an expanded version of this example, see the discussion of the **#if** directive in the section entitled "The #if, #elif, #else, and #endif Directives")

```
#define REG1        register
#define REG2        register
#define REG3
```

Example 4

This example defines a macro named *MAX*. Each occurrence of the identifier *MAX* after the definition in the source file is replaced by the expression

```
((x) > (y)) ? (x) : (y)
```

where actual values replace the parameters *x* and *y*. For example, the occurrence

```
MAX(1,2)
```

is replaced by

```
((1) > (2)) ? (1) : (2)
```

and the occurrence

```
MAX(i,s[i])
```

is replaced by

```
((i) > (s[i])) ? (i) : (s[i])
```

```
#define MAX(x,y)    ((x) > (y)) ? (x) : (y)
```

Because this macro is easier to read than the corresponding expression, the source program is easier to understand.

Note that arguments with side effects may cause this macro to produce unexpected results. For example, the occurrence *MAX(i, s[i++])* is replaced by *((i) > (s[i++])) ? (i) : (s[i++])*. The expression *(s[i++])* may be evaluated twice, so by the time the ternary expression has been fully evaluated, *i* will have been incremented either once or twice, depending on the result of the comparison.

Example 5

This example defines the macro *MULT*. Once the macro is defined, an occurrence such as *MULT(3, 5)* is replaced by *(3) * (5)*. The parentheses around the parameters are important because they control the interpretation when complex expressions form the arguments to the macro. For instance, the occurrence *MULT(3 + 4, 5 + 6)* is replaced by *(3 + 4) * (5 + 6)*, which evaluates to 77. Without the parentheses, the result would be *3 + 4 * 5 + 6*. This result evaluates to 29 because the multiplication operator (*) has higher precedence than the addition operator (+).

```
#define MULT(a,b)    ((a) * (b))
```

Example 6

This example defines two macros, one an object-like macro that expands to the string literal *Hello,World!*, and the other a function-like macro called *show*, which takes one argument. However, the definition of the second macro includes the stringizing operator (#) immediately preceding the formal parameter *x*. When an argument is passed to the *show* macro, the formal parameter is replaced by the actual argument enclosed in double quotation marks, thus ‘‘stringizing’’ it.

```
#define GREETING Hello, World!
#define show(x) printf(#x)

main()
{
        show(  x + z );
        printf("\n");
        show(n /* some comment */ + p);
        printf("\n");
        show(GREETING);  /* GREETING is not expanded; */
        printf("\n");    /* it is stringized instead */
        show('\x');
        printf("\n");
}
```

As the preprocessor progresses through the source file, the references to _show_ are expanded as follows:

show(x + z); produces _printf("x + z");_

show(n /* comment */ + p); produces _printf("n + p");_

show(GREETING); produces _printf("GREETING");_

and finally, _show('\x');_ produces _printf("'\\x' ");_

When the program is run, the screen output would be:

```
x + z
n + p
GREETING
'\x'
```

Example 7

This example illustrates use of both the ‘‘stringizing’’ and ‘‘token-pasting’’ operators in specifying program output.

```
#define paster(n) printf("token" #n " = %d", token##n)
```

If _token9_ is declared, and the macro is called with a numeric argument like:

```
paster(9) ;
```

the macro yields:

```
printf("token" "9" " = %d", token9) ;
```

which becomes

```
printf("token9 = %d", token9) ;
```

The #undef Directive

Syntax

#undef *identifier*

The **#undef** directive removes the current definition of *identifier*. Consequently, subsequent occurrences of *identifier* are ignored by the preprocessor. To remove a macro definition using **#undef**, give only the macro *identifier*; do not give a parameter list.

You can also apply the **#undef** directive to an identifier that has no previous definition. This ensures that the identifier is undefined.

The **#undef** directive is typically paired with a **#define** directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The **#undef** directive also works with the **#if** directive (see the section entitled ''The #if, #elif, #else, and #endif Directives'' later in this chapter.) to control conditional compilation of the source program.

Example

In this example, the **#undef** directive removes definitions of a manifest constant and a macro. Note that only the identifier of the macro is given.

```
#define WIDTH           80
#define ADD(X,Y)        (X) + (Y)
.
.
.
#undef WIDTH
#undef ADD
```

Include Files

Syntax

#include "*path-spec*"
#include <*path-spec*>

The **#include** directive adds the contents of a given ‘‘include file’’ to another file. You can organize constant and macro definitions into include files and then use **#include** directives to add these definitions to any source file. Include files are also useful for incorporating declarations of external variables and complex data types. You only need to define and name the types once in an include file created for that purpose.

The **#include** directive tells the preprocessor to treat the contents of the named file as if they appeared in the source program at the point where the directive appears. The new text can also contain preprocessor directives. The preprocessor carries out directives in the new text, then continues processing the original text of the source file.

The *path-spec* is a file name optionally preceded by a directory specification. It must name an existing file. The syntax of the file specification depends on the operating system the program is compiled on.

The preprocessor uses the concept of a ‘‘standard’’ directory or directories to search for include files. The location of the standard directories for include files depends on the implementation and the operating system. For a definition of the standard directories, see your compiler guide.

The preprocessor stops searching as soon as it finds a file with the given name. If you specify a complete, unambiguous path specification for the include file, between two sets of double quotation marks (‘‘ ’’), the preprocessor searches only that path specification and ignores the standard directories.

If the *path-spec* enclosed in double quotation marks is an incomplete path specification, the preprocessor first searches the ‘‘parent’’ file’s directory. A parent file is the file containing the **#include** directive. For example, if you include a file named *file2* within a file named *file1*, *file1* is the parent file.

Include files can be ‘‘nested,’’ that is, an **#include** directive can appear in a file named by another **#include** directive. For example, *file2*, above, could include *file3*. In this case, *file1* would still be the parent of *file2*, but would be the ‘‘grandparent’’ of *file3*.

When include files are nested, directory searching begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source currently being processed. If the file is not found, the search moves to directories specified on the compiler command line. Finally, the standard directories are searched.

If the file specification is enclosed in angle brackets, the preprocessor does not search the current working directory. It begins by searching for the file in the directories specified on the compiler command line, then in the standard directories.

Nesting of include files can continue up to 10 levels. Once the nested **#include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Example 1

This example adds the contents of the file named *stdio.h* to the source program. The angle brackets cause the preprocessor to search the standard directories for *stdio.h*, after searching directories specified on the command line.

```
#include <stdio.h>
```

Example 2

This example adds the contents of the file specified by *defs.h* to the source program. The double quotation marks mean that the preprocessor searches the directory containing the ‘‘parent’’ source file first.

```
#include "defs.h"
```

Conditional Compilation

This section describes the syntax and use of directives that control "conditional compilation." These directives let you suppress compilation of parts of a source file by testing a constant expression or identifier to determine which text blocks will be passed on to the compiler and which text blocks will be removed from the source file during preprocessing.

The #if, #elif, #else, and #endif Directives

Syntax

#if *restricted-constant-expression*
  [*text-block*]
[**#elif** *restricted-constant-expression*
  text-block]
[**#elif** *restricted-constant-expression*
  text-block]
.
.
.
[**#else**
  text-block]
#endif

The **#if** directive, together with the **#elif**, **#else**, and **#endif** directives, controls compilation of portions of a source file. Each **#if** directive in a source file must be matched by a closing **#endif** directive. Any number of **#elif** directives can appear between the **#if** and **#endif** directives, but at most one **#else** directive is allowed. The **#else** directive, if present, must be the last directive before **#endif**.

The preprocessor selects one of the given occurrences of *text-block* for further processing. A block specified in *text-block* can be any sequence of text. It can occupy more than one line. Usually *text-block* is program text that has meaning to the compiler or the preprocessor.

The preprocessor processes the selected *text-block* and passes it to the compiler. If *text-block* contains preprocessor directives, the preprocessor carries out those directives.

Any text blocks not selected by the preprocessor are removed from the file during preprocessing. Thus, these text blocks are not compiled.

The preprocessor selects a single *text-block* by evaluating the restricted constant expression following each **#if** or **#elif** directive until it finds a true (nonzero) restricted constant expression. It selects all text (including other preprocessor directives beginning with #) up to its associated **#elif**, **#else**, or **#endif**.

If all occurrences of *restricted-constant-expression* are false, or if no **#elif** directives appear, the preprocessor selects the text block after the **#else** clause. If the **#else** clause is omitted, and all instances of *restricted-constant-expression* in the **#if** block are false, no text block is selected.

Each *restricted-constant-expression* follows the rules for restricted constant expressions discussed in the section entitled ''Constant Expressions'' in Chapter 5. Such expressions cannot contain **sizeof** expressions, type casts, or enumeration constants. However, they can contain the preprocessor operator **defined** in special constant expressions, as shown by the following syntax:

defined(*identifier***)**

This constant expression is considered true (nonzero) if the *identifier* is currently defined; otherwise, the condition is false (0). An identifier defined as empty text is considered defined.

The **#if**, **#elif**, **#else**, and **#endif** directives can nest in the text portions of other **#if** directives. Each nested **#else**, **#elif**, or **#endif** directive belongs to the closest preceding **#if** directive.

Example 1

In this example, the **#if** and **#endif** directives control compilation of one of three function calls. The function call to *credit* is compiled if the identifier *CREDIT* is defined. If the identifier *DEBIT* is defined, the function call to *debit* is compiled. If neither identifier is defined, the call to *printerror* is compiled. Note that *CREDIT* and *credit* are distinct identifiers in C because their cases are different.

```
#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif
```

Example 2

Examples 2 and 3 assume a previously defined manifest constant named *DLEVEL.*

Example 2 shows two sets of nested **#if**, **#else**, and **#endif** directives. The first set of directives is processed only if *DLEVEL > 5* is true. Otherwise, the second set is processed.

```
#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
              #define STACK   200
    #else
              #define STACK   100
    #endif
#else
    #define SIGNAL  0
    #if STACKUSE == 1
              #define STACK   100
    #else
              #define STACK   50
    #endif
#endif
```

Example 3

In Example 3, **#elif** and **#else** directives are used to make one of four choices, based on the value of *DLEVEL.* The manifest constant *STACK* is set to 0, 100, or 200, depending on the definition of *DLEVEL.* If *DLEVEL* is greater than 5, *display(debugptr);* is compiled and *STACK* is not defined.

```
#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif
```

Example 4

Example 4 uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns register storage to variables in the order in which the **register** declarations appear in the source file. If a program contains more **register** declarations than the machine allows, the compiler honors earlier declarations over later ones. The program may be less efficient if the variables declared later are more heavily used.

```
#define REG1    register
#define REG2    register

#if defined(M_86)
    #define REG3
    #define REG4
    #define REG5
#else
    #define REG3    register
    #if defined(M_68000)
         #define REG4    register
         #define REG5    register
    #else
         #define REG4    register
         #define REG5
    #endif
#endif
```

The definitions listed in Example 4 can be used to give priority to the most important register declarations. *REG1* and *REG2* are defined as the **register** keyword to declare **register** storage for the two most important variables in the program. For example, in the following fragment, *b* and *c* have higher priority than *a* or *d*:

```
func(a)

REG3 int a;

{
        REG1 int b;
        REG2 int c;
        REG4 int d;
        .
        .
        .
}
```

When *M_86* is defined, the preprocessor removes the *REG3* identifier from the file by replacing it with empty text. This prevents *a* from receiving **register** storage at the expense of *b* and *c*. When *M_68000* is defined, all four variables are declared to have **register** storage. When neither *M_86* nor *M_68000* is defined, *a*, *b*, and *c* are declared with **register** storage.

The #ifdef and #ifndef Directives

Syntax

#ifdef *identifier*
#ifndef *identifier*

The **#ifdef** and **#ifndef** directives perform the same task as the **#if** directive used with **defined(***identifier***)**. You can use the **#ifdef** and **#ifndef** directives anywhere **#if** can be used. These directives are provided only for compatibility with previous versions of the language. The **defined(***identifier***)** constant expression used with the **#if** directive is preferred.

When the preprocessor encounters an **#ifdef** directive, it checks to see whether the *identifier* is currently defined. If so, the condition is true (nonzero); otherwise, the condition is false (0).

The **#ifndef** directive checks for the opposite of the condition checked by **#ifdef**. If the identifier has not been defined (or its definition has been removed with **#undef**), the condition is true (nonzero). Otherwise, the condition is false (0).

Line Control

Syntax

#line *constant* ["*filename*"]

The **#line** directive tells the preprocessor to change the compiler's internally stored line number and file name to a given line number and file name. The compiler uses the line number and file name to refer to errors that it finds during compilation. The line number normally refers to the current input line, and the file name refers to the current input file. The line number is incremented after each line is processed.

If you change the line number and file name, the compiler ignores the previous values and continues processing with the new values. The **#line** directive is typically used by program generators to cause error messages to refer to the original source file instead of to the generated program.

The *constant* value in the **#line** directive can be any integer constant. The *filename* can be any combination of characters and must be enclosed in double quotation marks (" "). If *filename* is omitted, the previous file name remains unchanged.

The current line number and file name are always available through the predefined identifiers __**LINE**__ and __**FILE**__. You can use the __**LINE**__ and __**FILE**__ identifiers to insert self-descriptive error messages into the program text.

The __**FILE**__ identifier expands to a string whose contents are the file name, surrounded by double quotation marks (" ").

Example 1

In this example, the internally stored line number is set to 151 and the file name is changed to *copy.c*.

```
#line 151 "copy.c"
```

Example 2

In this example code segment , the macro *ASSERT* uses the predefined identifiers **__LINE__** and **__FILE__** to print an error message about the source file if a given "assertion" is not true.

```
#define ASSERT(cond)          if(!cond)\
{printf("assertion error line %d, file(%s)\n", \
__LINE__, __FILE__ );} else
printf("condition true");
```

Pragmas

Syntax

#pragma *character-sequence*

A **#pragma** is an implementation-defined instruction to the compiler. The *character-sequence* is a series of characters that gives a specific compiler instruction and arguments, if any. The number sign (**#**) must be the first non-white-space character on the line containing the pragma; white-space characters can separate the number sign and the word **pragma**.

See your compiler guide for information about the pragmas available in your compiler implementation.

Appendix A

Differences Between K&R C and Microsoft C

Introduction A-1

Introduction

This appendix summarizes differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published in 1978 by Prentice-Hall, Inc. The following is a list of the differences with cross-references to the corresponding section numbers in *The C Programming Language*:

Section Number in Kernighan and Ritchie	Microsoft C
2.2	Identifiers (including those used in preprocessor directives) are significant to 31 characters. External identifiers are also significant to 31 characters.
2.3	The identifiers **asm** and **entry** are no longer keywords. New keywords are **const**, **volatile**, **enum**, **signed**, and **void**. (The **volatile** keyword is implemented syntactically, but not semantically.) The identifiers **cdecl**, **far**, **fortran**, **huge**, **near**, and **pascal** may be keywords, depending on whether the corresponding options are enabled when a program is compiled (see your system documentation).
2.4.1	As a result of the method used to assign types to hexadecimal and octal constants, these constants always act like unsigned integers in type conversions.
2.4.3	Hexadecimal bit patterns consisting of a backslash (\), the letter **x**, and up to three hexadecimal digits are permitted as character constants (for example, \x012).
	Microsoft C defines three additional escape sequences: **\v** represents a vertical tab (VT), **\"** represents the double-quotation-mark character, and **\a** represents the bell (also called alert).
	Character constants always have type **int**, with the result that they are sign extended in type conversions.
	Adjacent quoted string literals are concatenated and treated as a single null-terminated string.

2.6 The **short** type is always 16 bits long, and the **long** type is 32 bits long. The size of an **int** is machine dependent. On 8086/8088, 80186, and 80286 processors an **int** is 16 bits long, and on 80386 and 68000 processors it is 32 bits long.

4 The **char** type is signed by default, with the result that a **char** value is sign extended in type conversions. (In some implementations, the default for the **char** type can be changed to unsigned at compile time.)

Two additional unsigned types are supported: **unsigned char** and **unsigned long**.

The keyword **unsigned** or **signed** can be applied as an adjective to an integer type. When **unsigned** appears alone, it means **unsigned int**. Similarly, when **signed** appears alone, it means **int**. The additional floating type **long double** is supported, but the **long float** type is no longer recognized. References to **long float** should be recoded to **double**.

The type specifiers **const** and **volatile** can be used as modifiers for any fundamental, aggregate, or pointer type. The **const** keyword indicates that the object or pointer value will not be modified. The **volatile** keyword means the object may be changed by some process beyond the control of the currently running program. Both the syntax and semantics of **const** are implemented, but only the syntax of **volatile** is implemented.

Microsoft C offers an additional fundamental type: the **enum** (enumeration) type. Variables of **enum** type are treated as integers in all cases.

The keyword **void** has three different uses. As a function-return-type specifier, it indicates that the function will not return a value. In an otherwise empty formal-parameter list, **void** means that no arguments will be passed. In the construction **void ***, it indicates a pointer to an object of unspecified type.

6.4 If the **near**, **far**, and **huge** keywords are enabled, pointers of different sizes can be used in a program. Operations with pointers of different sizes can cause conversion of pointers; the path of the conversion is implementation defined.

6.6 Arithmetic conversions carried out by the compiler are outlined in the ‘‘Expressions and Assignments’’ chapter. Although compatible with the Kernighan and Ritchie conversions, Microsoft C conversions are described in greater detail, including the specific path for each type of conversion.

In addition to the usual arithmetic conversions, conversions between pointers of different sizes can be routinely carried out when the **near**, **far**, and **huge** keywords are enabled. The path of the pointer conversions is implementation defined.

7.2 In connection with the **sizeof** operator, a byte is defined as an 8-bit quantity.

7.14 A structure can be assigned to another structure of the same type.

8.2 The keywords **enum**, **const**, **volatile**, and **void** are additional type specifiers. The **volatile** keyword is implemented syntactically, but not semantically. The keywords **signed** and **unsigned** can serve either as type specifiers or as adjectives modifying an integral type.

Therefore, the following additional combinations are acceptable:

signed char
signed short
signed short int
signed long
signed long int
unsigned char
unsigned short
unsigned short int
unsigned long
unsigned long int

The **long float** type is not recognized. The **long double** type is recognized and treated in all instances the same as **double.**

8.4 The **const** and **volatile** keywords can be used to modify any fundamental, aggregate, or pointer object. The order of the type specifiers is not significant.

Optional formal-parameter lists or argument-type lists can be included in function declarations to notify the compiler of the number and types of arguments expected in a function call.

8.5 Bitfields can be declared to be any **signed** or **unsigned** integral type, except **enum**. However, in expressions, bitfields are always treated as **unsigned**.

The names of structure and union members are not required to be distinct from structure and union tags or from the names of other variables.

No relationship exists between the members of two different structure types.

8.6 Unions can be initialized by giving a value for the first member of the union.

9.7 The *expression* of a **switch** can be any integral expression, but the value of the expression is always converted to an **int** type. An **enum** type is permitted for *expression.* Each of the **case** constant expressions is cast to the type of *expression.*

10.1 New styles for function declarations and definitions, as specified in the Draft Proposed American National Standard — Programming Language C, are completely supported. This includes the function prototype declaration, the prototype-style definition with formal parameters declared in the header, and the default creation of prototypes from the first reference to a function (if no explicit prototype is provided). The old function declaration and definition forms are also supported.

The formal parameter list in a function definition or declaration can end with a comma followed by three periods (,...) or just a comma (,) to indicate that the number of parameters is variable. The latter is supported only for compatibility with older versions of the compiler and should not be used in new code.

12 The number sign (#) introducing the preprocessor directive can be preceded by any combination of white-space characters. White space can also separate the number sign and the preprocessor keyword.

In addition to preprocessor directives, the source file can contain pragmas. Pragmas, like directives, are introduced by a number sign as the first non-white-space character in a line. The action defined by a particular pragma is implementation dependent.

Three preprocessor-only operators are supported: the ‘‘stringizing’’ operator (#), the concatenation or ‘‘token-pasting’’ operator (##), and the **defined** operator.

12.3 The new combination **#if defined** (*identifier*) is intended to supplant the **#ifdef** and **#ifndef** directives. Use of the latter directives is discouraged.

The new directive **#elif** (else if) is designed for use in **#if** and **#if defined** blocks.

14.1 A structure or union can be assigned to another structure or union of the same type. Structures and unions can be passed by value to functions and returned by functions.

In expressions involving the structure-pointer operator (–>), the expression preceding the arrow must have the same type (or must be cast to the same type) as the structure to which the member on the right-hand side of the arrow belongs.

17 The listed anachronisms are not recognized.

Appendix B

Syntax Summary

Tokens B-1
  - Keywords B-1
  - Identifiers B-2
  - Constants B-2
  - Strings B-5
  - Operators B-6
  - Separators B-6

Expressions B-7

Declarations B-9

Statements B-14

Definitions B-15

Preprocessor Directives B-16

Pragmas B-17

Tokens

keyword
identifier
constant
string
operator
separator

Keywords

auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile†
while

---

† Semantics not yet implemented

The following identifiers may be keywords in some implementations. For information, see your compiler guide.

cdecl
far
fortran
huge
near
pascal

Identifiers

identifier:
  letter
  underscore
  identifier letter
  identifier underscore
  identifier digit

letter—one of the following:
  a b c d e f g h i j k l m
  n o p q r s t u v w x y z
  A B C D E F G H I J K L M
  N O P Q R S T U V W X Y Z

underscore:
  _

digit—one of the following:
  0 1 2 3 4 5 6 7 8 9

Constants

constant:
  integer-constant
  long-constant
  floating-point-constant
  char-constant
  enum-constant

integer-constant:
 0
 decimal-constant
 octal-constant
 hexadecimal-constant

decimal-constant:
 nonzero-digit
 decimal-constant digit

nonzero-digit—one of the following:
 1 2 3 4 5 6 7 8 9

octal-constant:
 0*octal-digit*
 octal-constant octal-digit

octal-digit—one of the following:
 0 1 2 3 4 5 6 7

hexadecimal-constant:
 0x*hexadecimal-digit*
 0X*hexadecimal-digit*
 hexadecimal-constant hexadecimal-digit

hexadecimal-digit—one of the following:
 0 1 2 3 4 5 6 7 8 9
 a b c d e f
 A B C D E F

long-constant:
 integer-constant **l**
 integer-constant **L**

floating-point-constant:
  fractional-constant exponent
  fractional-constant
  digit-seq exponent

fractional-constant:
  digit-seq **.** *digit-seq*
  cc+
  +cc
  digit-seq **.**

digit-seq:
  digit
  digit-seq digit

exponent:
  e *sign digit-seq*
  E *sign digit-seq*
  e *digit-seq*
  E *digit-seq*

sign:
  +
  -

char-constant:
  '*char***'**

char:
  rep-char
  escape-sequence

rep-char:

Any single representable character except the single quotation-mark ('), backslash (\), or new-line character. Note that the single-quotation-mark character cannot be used alone in a character constant, and the double quotation-mark character cannot be used alone in a string literal.

escape-sequence—one of the following:

\' \" \\ \d \dd \ddd
\xd \xdd \xddd \a \b \f
\n \r \t \v

enum-constant:
identifier

Strings

string-literal:
""
"*char-seq*"

char-seq:
char
char-seq char

Operators

operator—one of the following:

```
!     ~     ++    --    +
-     *     /     %     <<
>>    <     <=    >     >=
==    !=    |     &     ^
&&    ||    =     +=    -=
=     /=    %=    >>=   <<=
&=    ^=    |=    ?:    ,
[ ]   ( )   .     ->
```

Separators

separator—one of the following:

```
[   ]   (   )   {   }
*   ,   :   =   ;   #
```

Expressions

expression:
identifier
constant
string
expression(*expression-list*)
expression(**void**)
expression[*expression*]
expression.*identifier*
expression->*identifier*
unary-expression
binary-expression
ternary-expression
assignment-expression
(*expression*)
(*type-name*)*expression*
constant-expression

expression-list:
expression
expression-list , *expression*

unary-expression:
unop expression
sizeof(*expression*)

unop—one of the following:
- ~ ! * &

lvalue:
identifier
expression[*expression*]
expression.*expression*
expression->*expression*
*expression
(*type-name*)*expression*
(*lvalue*)

Expressions

type-name:
See Section B.3, ‘‘Declarations.’’

binary-expression:
expression binop expression

binop—one of the following:

*   /   %   +   -
<<   >>   <   >   <=
>=   ==   !=   & |
^   && || ,

ternary-expression:
expression **?** *expression* **:** *expression*

assignment-expression:
lvalue++
lvalue--
++*lvalue*
--*lvalue*
lvalue assignment-op expression

assignment-op—one of the following:

=   *=   /=   %=   +=   -=
<<=   >>=   &=   /=   ^=

constant-expression:
identifier
constant
(*type-name*)*constant-expression*
unary-expression
binary-expression
ternary-expression
(*constant-expression*)

Declarations

declaration:
sc-specifier type-specifier-list declarator-list;
type-specifier-list declarator-list;
sc-specifier declarator-list;
typedef *type-specifier-list declarator-list*;

sc-specifier:
auto
extern
register
static

type-specifier:
char
double
long double
enum-specifier
float
int
long
short
struct-specifier
typedef-name
union-specifier
unsigned
signed
const
volatile
void

type-specifier-list:
type-specifier
type-specifier-list type-specifier

enum-specifier:
  enum *tag* {*enum-list*}
  enum {*enum-list*}
  enum *tag*

tag:
  identifier

enum-list:
  enumerator
  enum-list , *enumerator*

enumerator:
  identifier
  identifier = *constant-expression*

struct-specifier:
  struct *tag* {*member-declaration-list*}
  struct {*member-declaration-list*}
  struct *tag*

member-declaration-list:
  member-declaration
  member-declaration-list member-declaration

member-declaration:
  type-specifier declarator-list;
  type-specifier identifier : *constant-expression*;
  type-specifier : *constant-expression*;

declarator-list:
  declarator
  declarator = *initializer*
  declarator-list , *declarator*

declarator:
identifier
modifier-list identifier
declarator[ ]
declarator[*constant-expression*]
**declarator*
declarator(**void**)
declarator([*formal-parameter-list*])
(*declarator*)

modifier-list
modifier
modifier-list modifier

formal-parameter-list
formal-parameter
formal-parameter-list, *formal-parameter*
formal-parameter-list, . . .
formal-parameter-list,
formal-parameter
sc-spec type-spec declarator
sc-spec type-spec abstract-declarator

arg-type-list:
type-name
arg-type-list, *type-name*
arg-type-list, . . .
arg-type-list,

type-name:
type-specifier
type-specifier abstract-declarator

abstract-declarator:
*
*modifier**
[ ]
(*arg-type-list*)
**abstract-declarator*
*abstract-declarator**
abstract-declarator[ ]
abstract-declarator[*constant-expression*]
[ ]*abstract-declarator*
[*constant-expression*]*abstract-declarator*
abstract-declarator(**void**)
abstract-declarator(*formal-parameter-list*)
abstract-declarator(*arg-type-list*)
(*abstract-declarator*)

initializer:
expression
{*initializer-list*}

initializer-list:
initializer
initializer-list, *initializer*

typedef-name:
identifier

union-specifier:
union *tag* {*member-declaration-list*}
union {*member-declaration-list*}
union *tag*

modifier:
cdecl
far
fortran
huge
near
pascal

modifier-list
  modifier
  modifier-list modifier

Statements

statement:
break;
case *constant-expression* **:** *statement*
compound-statement
continue;
default : *statement*
do *statement* **while**(*expression*);
expression;
for ([*expression*];[*expression*];[*expression*]) *statement*;
goto *identifier*;
identifier **:** *statement*
if (*expression*) *statement* [**else** *statement*];
return [*expression*];
switch (*expression*) *statement*
while (*expression*) *statement*

compound-statement:
{[*declaration-list*][*statement-list*]}

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

Definitions

definition:
    function-definition
    data-definition

function-definition:
    [*sc-specifier*] [*type-specifier*] *declarator* ([*formal-parameter-list*])
        compound-statement
    [*sc-specifier*] [*type-specifier*] *declarator* ([*parameter-list*])
        [*parameter-decs*] *compound-statement*

parameter-list:
    fixed-parameter-list
    variable-parameter-list

fixed-parameter-list:
    identifier
    parameter-list , *identifier*

variable-parameter-list:
    fixed-parameter-list, ...
    fixed-parameter-list,

parameter-decs:
    declaration
    declaration-list declaration

data-definition:
    declaration

Preprocessor Directives

directive:

#define *identifier* [([*parameter-list*])][*token-seq*]
#elif *restricted-constant-expression*
#else
#endif
#if *restricted-constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#include "*string*"
#include <*string*>
#line *digit-seq*
#line *digit-seq string*
#undef *identifier*

token-seq:

token
token-seq token

restricted-constant-expression:

defined (*identifier*)
Any *constant-expression* except **sizeof** expressions, casts, and enumeration constants

Pragmas

pragma:
    #pragma *char-seq*

Index

Special Characters

- >> (right-shift operator) 5-24
- <> (angle brackets) 8-12
- -> (arrow), in member-selection expressions 5-7
- > (greater-than operator) 5-25
- >= (greater-than-or-equal-to operator) 5-25
- -> (member-selection operator) 5-7, A-5
- { } (braces) 4-50, 6-1, 6-4
- [ ] (brackets) 1-7
- [ ] (brackets)
  - array declarators, used in 4-9, 4-26
- ? : (conditional operator) 5-30
- ( ) (function modifier) 4-9
- ? : (ternary operator) 5-14, 5-30
- # # (token-pasting operator)
  - described 8-3, 8-6
  - differences from Kernighan and Ritchie 8-1
- + (addition operator) 5-22
- & (address-of operator) 5-18
- - (arithmetic negation operator) 5-16
- \ (backslash character) 2-5, 2-6
- & (bitwise-AND operator) 5-27
- ~ (bitwise-complement operator) 5-16
- ^ (bitwise-exclusive-OR operator) 5-27
- | (bitwise-inclusive-OR operator) 5-27
- [ ] (brackets)
  - subscript expressions, used in 5-4, 5-6
- : (colon), with bitfield structure members 4-22
- , (comma)
  - argument-type lists, used in 4-35
  - declarations, used in 4-17, 4-33
  - function calls, used in 5-3, 7-16
  - initialization, used in 4-50
  - sequential-evaluation operator 5-29
- -- (decrement operator) 5-34
- / (division operator) 5-21
- ... (ellipsis notation) 4-35
- == (equality operator) 5-25
- ++ (increment operator) 5-34
- * (indirection operator) 5-17
- != (inequality operator) 5-25
- << (left-shift operator) 5-24
- < (less-than operator) 5-25
- <= (less-than-or-equal-to operator) 5-25
- && (logical-AND operator) 5-28
- ! (logical-NOT operator) 5-16
- || (logical-OR operator) 5-28
- . (member-selection operator) 5-7
- * (multiplication operator) 5-21
- # (number sign) 8-1
- ~ (one's complement operator) 5-16
- ( ) (parentheses)
  - complex declarators, used in 4-10
  - expressions, used in 5-10
  - function calls, used in 5-3
  - function declarators, used in 4-9, 4-33
  - macros, used in 8-9
- * (pointer modifier) 4-9, 4-28
- % (remainder operator) 5-21
- = (simple-assignment operator) 5-35
- # (stringizing preprocessor operator) 8-3
- - (subtraction operator) 5-23
- - (two's complement operator) 5-16
- + (unary plus operator) 5-16
- _ (underscore character) 2-17

A

- Abstract declarators 4-58
- Actual arguments *See* Arguments, actual
- Addition operator (+) 5-22
- Address-of operator (&) 5-18
- Aggregate data-type category 4-7
- Aggregate types
  - array 4-26
  - initialization 4-48, 4-50
  - structure 4-21
  - union 4-25
- Anachronisms A-5
- AND operators
  - bitwise (&) 5-27
  - logical (&&) 5-28
- Angle brackets (<>) 8-12
- ANSI standard
  - enabling ANSI 1-1
  - extensions 1-1
- Apostrophe (') *See* Escape sequences
- argc parameter 3-7
- Argument type checking
  - conversions 7-20
  - default prototypes 7-14
  - formal parameters 7-9
  - function calls 7-20
  - variable-length parameter list 4-36

Arguments
  actual
    conversion 7-20
    evaluation, order of 7-17
    macros 8-5, 8-9
    passing 7-19
    pointers 7-17, 7-20
    side effects 7-17
    type checking 7-20
    variable number 7-22
  command line 3-7
  formal *See* Formal parameters
  main function 3-7
  variable number 4-35, 7-22
Argument-type lists
  abstract declarator, used with 4-58
  default prototype 7-14
  described 4-34
  pointer arguments, used with 4-36
  variable length 4-35
  void *, used with 4-36
  void keyword, used with 4-36
argv parameter 3-7
Arithmetic conversions 5-15, A-3
Arithmetic data-type category 4-7
Arithmetic negation operator (-) 5-16
Array declarators ([ ]) 4-9, 4-26
Arrays
  declarations 4-9, 4-26
  elements 5-4
  identifiers 5-2
  initialization 4-48, 4-50, 4-54
  multidimensional 4-27, 5-6
  references to 5-2, 5-4
  storage 4-27, 5-6
  subscripts 5-4
asm keyword A-1
Assignments
  conversions 5-41
  defined 5-1
  expressions 5-9
  operators 5-32
Associativity
  modifiers 4-10
  operators 5-37
auto storage class 4-40, 4-44, 4-48

B

Backslash character (\) 2-5, 2-6
Backspace escape sequence (\b) 2-5
Bell character (\a) 2-5, A-1
Binary expressions 5-9
Binary operators, table 2-7, 5-14
Bitfields 4-22, 4-23, A-4
Bitwise-AND operator (&) 5-27
Bitwise-complement operator (~) 5-16
Bitwise-exclusive-OR operator (^) 5-27
Bitwise-inclusive-OR operator (|) 5-27
Blocks 3-8
Braces ({ })
  compound statement, used in 6-1, 6-4
  initialization, used in 4-50
Brackets
  array declarators, used in 4-9, 4-26
  subscript expressions, used in 5-4, 5-6
Brackets ([ ]) 1-7
Branch statements 6-13, 6-18
break statement 6-3
Bytes, size of A-3

C

C character set 2-2
Call by reference *See* Passing by reference
Call by value *See* Passing by value
Calls *See* Function calls
Carriage-return escape sequence (\r) 2-5
case keyword 6-18
Case sensitivity 2-3, 2-17
Casts *See* type casts
cdecl keyword 2-19, 4-14, A-1
char type
  conversion 5-42
  described 4-2
  differences from Kernighan and Ritchie A-2
  range of values 4-4
  storage 4-4
Character constants
  differences from Kernighan and Ritchie A-1
  form 2-13
  sign extension 2-14
  type 2-14
Character sets 2-2
Characters
  backslash (\) 2-5, 2-6
  backspace escape sequence 2-5
  bell (\a) 2-5, A-1

Characters *(continued)*
  carriage-return escape sequence (\r) 2-5
  case 2-3, 2-17
  continuation (\) 2-6
  differences from Kernighan and Ritchie A-1
  digits 2-3
  double-quotation-mark escape sequence 2-5
  escape sequences 2-5
  form-feed escape sequence (\f) 2-5
  hexadecimal escape sequences 2-5
  horizontal tab escape sequence (\t) 2-5
  letters 2-3
  new-line escape sequence (\n) 2-5
  octal escape sequences 2-5
  punctuation 2-4
  single-quotation-mark escape sequence (\') 2-5
  special 2-4
  underscore (_) 2-3
  vertical-tab escape sequence (\v) 2-5
  white space 2-3, 2-5
Colon (:), with bitfield structure members 4-22
Comma (,)
  argument-type lists, used in 4-35
  declarations, used in 4-17, 4-33
  function calls, used in 5-3, 7-16
  initialization, used in 4-50
  sequential-evaluation operator (,) 5-29
Command-line arguments 3-7
Comments 2-20
Compilation, conditional 8-14, 8-18
Complement operators (~) 5-16
Complex declarators 4-10, 4-14
Compound statements 6-4
Compound-assignment operators 5-36
Concatenation of string literals 2-15
Concatenation operator, differences from Kernighan and Ritchie A-5
Conditional compilation 8-14, 8-18
Conditional operator (? :) 5-30
Conditional statements 6-13, 6-18
const
  keyword A-1
  pointer modifier, used as 4-28
  type specifier 4-3
Constant expressions
  case 6-18
  conversion 4-7
  defined (identifier) 8-15
  described 5-1
  directives, used in 5-11, 8-15
  form 5-11
  initializers 5-11
  restricted 5-11, 8-15

Constant expressions *(continued)*
  switch statement, used in 6-18
Constants
  character *See* Character constants
  conversion 4-7
  decimal integer 2-10, 2-11
  described 2-10
  enumeration 4-20
  floating point 2-12, 4-6
  hexadecimal integer
    conversion 2-11, 4-7
    form 2-10
    type 2-11
  integer
    differences from Kernighan and Ritchie A-1
    form 2-10
    long 2-12
    negative 2-11
    octal *See* Octal constants
    type 2-11
  manifest 8-3, 8-4, 8-11
  string *See* String literals
  summarized B-2
  type 5-2
Continuation character (\) 2-6
continue statement 6-5
Control, returning 6-16
Conventions, notational 1-5
Conversions
  actual arguments 7-20
  assignment 5-41
  constant expressions 4-7
  constants 4-7
  enumeration types 5-49
  floating types 5-46
  formal parameters 7-9, 7-20
  function call 5-50, 7-20
  function prototypes 5-50
  hexadecimal constants 4-7
  implicit 5-49
  octal constants 4-7
  operator 5-50
  pointer types 5-48
  range of values, effects on 4-7
  signed integral types 5-41, 5-49
  structure types 5-49
  type cast 5-50
  union types 5-49
  unsigned integral types 5-44, 5-49
  usual arithmetic 5-15, A-3
  void type 5-49

D

- Data type categories 4-7
- Data types *See* Types
- Decimal integer constants 2-10, 2-11
- Declaration modifiers 4-14
- Declarations
  - defining 3-3
  - form 4-1
  - formal parameter names 4-33
  - formal parameters 7-7, 7-8
  - forward *See* Function declarations (prototypes)
  - function *See* Function declarations (prototypes)
  - pointer 4-9, 4-28, 7-14
  - referencing 3-3
  - storage allocation 3-3
  - summarized B-9
  - type 4-55
  - typedef 4-55, 4-56
  - variable
    - array 4-26
    - default storage class 4-42
    - described 3-2
    - enumeration 4-19
    - external 4-40
    - form 4-17
    - global 4-41
    - internal 4-40
    - local 4-44
    - multidimensional arrays 4-27
    - pointer 4-28
    - simple 4-18
    - structure 4-21
    - union 4-25
- Declarators
  - abstract 4-58
  - array 4-9
  - complex 4-10, 4-14
  - described 4-9
  - function 4-9
  - parentheses, enclosed in 4-10
  - pointer 4-9
  - special keywords, used with 4-14
- Decrement operator (--) 5-34
- default keyword 6-18
- Default return type 4-33
- Default storage class
  - function declarations 4-47
  - global variable declarations 4-42
  - local variable declarations 4-44
- #define directive 8-4
- defined (identifier) constant expression 8-15
- defined preprocessor operator 8-1, 8-3, A-5
- Defining declaration 4-41
- Definitions
  - function
    - described 3-3, 7-1, 7-3
    - full prototype form 7-3
    - obsolete form 7-4
    - storage class 7-4
    - summarized B-15
    - visibility 7-4
  - removing 8-11
  - storage allocation 3-3
  - variable
    - described 3-3, 4-41
    - storage class 4-41
    - summarized B-15
    - visibility 4-41, 4-44
- Digits 2-3
- Dimensions *See* Multidimensional arrays
- Directives
  - constant expressions, used in 5-11, 8-15
  - #define 8-4
  - described 3-2, 8-1
  - differences from Kernighan and Ritchie A-4, A-5
  - #elif
    - described 8-14
    - differences from Kernighan and Ritchie A-5
    - nesting 8-15
  - #else 8-14, 8-15
  - #endif 8-14, 8-15
  - #if 8-14, 8-15, A-5
  - #ifdef 8-18, A-5
  - #ifndef 8-18, A-5
  - #include 8-12
  - lifetime 3-5
  - #line 8-19
  - restricted constant expressions 5-11
  - summarized B-16
  - #undef 8-11
- Division operator (/) 5-21
- do statement
  - described 6-6
  - execution
    - continuation of 6-5
    - termination of 6-3
- Double quotation mark ( ) *See* Quotation marks
- double type
  - conversion 5-47
  - described 4-2
  - internal representation 4-6
  - range of values 4-4

double type *(continued)*
  range of values 4-4
  storage 4-4
Double-quotation-mark escape sequence *See* Escape sequences

E

Elements 5-4, 5-6
#elif directive
  described 8-14
  differences from Kernighan and Ritchie A-5
  nesting 8-15
Ellipsis notation (...) 1-6
#else directive 8-14, 8-15
else keyword 6-13
#endif directive 8-14, 8-15
entry keyword A-1
enum type specifier 4-19, A-1
Enumeration constants 3-14, 4-20
Enumeration expressions 5-2
Enumeration set 4-19
Enumeration types
  conversion 5-49
  declaration 4-19, 4-55
  described 4-2
  differences from Kernighan and Ritchie A-2
  identifiers 5-2
  range of values 4-4
  storage 4-4, 4-19
  tags
    defined 3-14
    naming class 3-14
    type declarations 4-55
    variable declarations 4-19
Enumeration variables 4-17
envp 3-7
Equality operator (==) 5-25
Escape sequences
  described 2-5
  differences from Kernighan and Ritchie A-1
  double quotation mark 2-5
  \' (single quotation mark) 2-5
  \a (bell) 2-5
  \b (backspace) 2-5
  \\ (backslash) 2-5
  \f (form feed) 2-5
  \n (new line) 2-5
  \r (carriage return) 2-5
  \t (horizontal tab) 2-5
  \v (vertical tab) 2-5
Evaluation
Evaluation *(continued)*
  order of 5-28, 5-38
  unary plus (+), forcing order with 5-16
Execution *See* Program execution
Exit from functions 6-16
Exponents 2-12
Expressions
  assignment 5-9
  binary 5-9
  case constant 6-18
  constant *See* constant expressions
  described 5-1
  enumeration 5-2
  floating type 5-2
  function call 5-4
  grouping 5-37
  integral 5-2
  list 5-3
  lvalue 5-33
  member selection 5-7, A-5
  operators, used in 5-9
  order of evaluation 5-38
  parentheses, enclosed in 5-10
  pointer 5-2
  side effects 5-12
  statements 6-7
  string literal 5-3
  structure 5-2
  subscript 5-4, 5-6
  summarized B-7
  switch 6-18, A-4
  ternary 5-9
  type cast 5-10
  unary 5-9
  union 5-2
Extensions to ANSI C standard 1-1
extern storage class
  described 4-40
  function
    declarations 4-46
    definitions 7-4
  function declarations 7-13
  global variables 4-41
  local variables 4-44
External declarations
  described 4-40
  function 4-46
External level 3-3

F

far keyword
  conversions 7-20
  described 4-14
  differences from Kernighan and Ritchie A-1
  listed 2-19
Fields *See* Bitfields
__FILE__ identifier 8-19
Files
  inclusion 8-12
  name, changing 8-19
  nesting 8-13
float type
  conversion 5-46
  described 4-2
  internal representation 4-6
  range of values 4-4
  storage 4-4
Floating point
  constants
    form 2-12
    internal representation 4-6
    negative 2-12
  data-type category 4-7
  expressions 5-2
  identifiers 5-2
  types
    described 4-2
    internal representation 4-6
  types, conversion of 5-46
for statement
  described 6-9
  execution continuation 6-5
  execution termination 6-3
Forcing evaluation order 5-16
Formal parameters
  conversion 7-9, 7-20
  declaration 7-8
  described 4-34, 7-7
  following function header 7-4
  identifiers 7-9
  list 7-3
  macro 8-4
  names 4-33
  naming class 3-13
  obsolete form 7-8
  storage class 7-9
  type checking 7-9, 7-20
Form-feed escape sequence (\f) 2-5
fortran keyword 2-19, 4-14, A-1
Forward declarations *See* Function declarations (prototypes)
Function
  body 7-4, 7-11
  calls
    argument type checking 7-20
    arguments,variable number of 7-22
    conversions 5-50, 7-20
    described 7-1
    expressions 5-4
    form 5-3, 7-16
    indirect 7-17
    operator, used as sequence point 5-12
    pointers, use of 7-17
    recursive 7-23
  declarations (prototypes)
    arguments, variable number of 4-35
    arguments, without 4-36
    default return type 4-33
    default storage class 4-47
    described 3-2, 7-1, 7-13
    differences from Kernighan and Ritchie A-4
    implicit 7-13
    parameter list 4-36
    pointer 4-33
    pointer arguments 4-36
    return type 4-34, 7-13
    return value 7-12
    storage class 4-46, 4-47, 7-13
    visibility 4-46, 7-13
  definition
    full prototype form 7-3
    obsolete form 7-4
  definitions *See* Definitions function
  modifier ( ) 4-9
  names *See* Identifiers
  pointers 7-14, 7-17
  prototypes
    conversions 5-50
    defined 4-36, 7-1
  return type *See* Return type
  type *See* Return type
Function-like macros 8-3
Functions
  described 7-1
  exit from 6-16
  identifiers 5-3
  main 3-7
  naming class 3-13
  return value 6-16

G

Global
  level 3-3
  lifetime 3-8, 4-40
  variables
    described 3-10
    initialization 4-48
    references to 4-45
  visibility 3-9
Global declarations
  variable 4-41
goto statement 6-11
Greater-than operator (>) 5-25
Greater-than-or-equal-to operator (>=) 5-25

H

Hexadecimal
  constants
    conversion 2-11, 4-7
    differences from Kernighan and Ritchie A-1
    form 2-10
    sign extension 2-11
    type 2-11
  escape sequences 2-5, A-1
Horizontal-tab escape sequence (\t) 2-5
huge keyword
  conversion 7-20
  described 4-14
  differences from Kernighan and Ritchie A-1
  listed 2-19

I

Identifier lists 7-8
Identifiers
  array 5-2
  characters allowed 2-17
  differences from Kernighan and Ritchie A-1
  enumeration 5-2
  __FILE__ 8-19
  floating type 5-2
  formal parameters 7-9
  function 5-3
  integral 5-2
  length 2-17
Identifiers *(continued)*
  __LINE__ 8-19
  modified 4-9
  naming classes 3-13
  pointer 5-2
  structure 5-2
  summarized B-2
  union 5-2
#if directive 8-14, 8-15, A-5
if statement 6-13
#ifdef directive 8-18, A-5
#ifndef directive 8-18, A-5
#include directive 8-12
Include files 8-12, 8-13
Increment operator (++) 5-34
Indirection operator (*) 5-17
Inequality operator (!=) 5-25
Initialization
  arrays 4-48, 4-50, 4-54
  auto storage class 4-48
  constant expressions 5-11
  differences from Kernighan and Ritchie A-4
  fundamental types 4-49
  global variables 4-48
  link time 4-42
  pointers 4-49
  register storage class 4-48
  restrictions 4-48
  static variables 4-48
  string literals 4-54
  structure variables 4-48, 4-50
  union variables 4-48, 4-50
Insertion of files 8-12
int type
  conversion 5-43
  described 4-2
  differences from Kernighan and Ritchie A-2
  portability 4-6
  range of values 4-4, 4-5
  storage 4-4
Integer constants
  decimal 2-10, 2-11
  differences from Kernighan and Ritchie A-1
  hexadecimal 2-10, 2-11
  long 2-12
  negative 2-11
  octal 2-10, 2-11
Integral
  data-type category 4-7
  expressions 5-2
  identifiers 5-2
  types
    conversion 5-41, 5-44, 5-49
    described 4-2

Internal
  declarations 4-40
  representation 4-6, 4-7
Internal level 3-3
Italics 1-5
Iterative statements
  do 6-6
  for 6-9
  while 6-21

K

Keywords
  differences from Kernighan and Ritchie A-1, A-3
  listed 2-19, B-1
  notational conventions 1-5
  special 4-14, 4-29
  statements, used in 6-1
  system dependent 2-19

L

Labeled statements 6-11
Labels
  case 6-18
  default 6-18
  described 6-1
  form 6-11
  naming class 3-14
Left-shift operator (<<) 5-24
Less-than operator (<) *See* Relational operators
Less-than-or-equal-to operator (<=) *See* Relational operators
Letters 2-3
Lifetime
  described 3-8
  directives 3-5
  global 3-8, 4-40
  local 3-8, 4-40
Line control 8-19
#line directive 8-19
__LINE__ identifier 8-19
Lines, continuation 2-6
Linked lists 4-22
_loadds keyword 4-14
Local
  declarations 4-44
  level 3-3
Local *(continued)*
  lifetime 3-8, 4-40
  variables 3-10, 7-12
Logical-AND operator (&&) 5-28
Logical-NOT operator (!) 5-16
Logical-OR operator (||) 5-28
long type
  conversion 5-43
  described 4-2
  differences from Kernighan and Ritchie A-2
  range of values 4-4
  storage 4-4
long-double type, conversion 5-47
long-float type 4-2
Loops
  do statement 6-6
  for statement 6-9
  while statement 6-21
Lvalue expressions 5-33

M

Macros
  actual arguments 8-5
  #define directive 8-4
  described 8-3
  empty definition 8-4
  example, with arguments 8-9
  example, with side effects 8-9
  function like 8-3
  object like 8-3
  side effects of arguments 8-5
  #undef, effect of 8-11
Main function 3-7
Manifest constants 8-3, 8-4, 8-11
Members
  bitfields 4-22
  naming class 3-14
  referring to 5-7
  structure 4-21
  union 4-25
Member-selection expressions 5-7, A-5
Member-selection operators (-> and .) 5-7, A-5
Modifiers
  array 4-9, 4-26
  associativity 4-10
  declaration 4-14
  function 4-9
  pointer 4-9, 4-28
  precedence 4-10
Multidimensional arrays 4-27, 5-6
Multiplication operator (*) 5-21

N

- Names *See* Identifiers
- Naming classes 3-13, A-4
- near keyword
  - conversions 7-20
  - described 4-14
  - differences from Kernighan and Ritchie A-1
  - listed 2-19
- Negation 5-16
- Nested visibility 3-10
- New-line escape sequence (\n) 2-5
- Nongraphic escape sequences 2-5
- NOT operator (!) *See* Logical-NOT operator
- Notational conventions 1-5
- Null statement 6-15
- Number sign (#) 8-1

O

- Object-like macros 8-3
- Octal
  - constants
    - conversion 2-11, 4-7
    - differences from Kernighan and Ritchie A-1
    - form 2-10
    - sign extension 2-11
    - type 2-11
  - escape sequences 2-5
- One's complement operator (~) 5-16
- Operands 5-1
- Operators
  - addition (+) 5-22
  - address of (&) 5-18
  - arithmetic negation (-) 5-16
  - assignment
    - compound 5-36
    - listed 5-32
    - simple (=) 5-35
  - associativity 5-37
  - binary
    - described 5-14
    - table 2-7
  - bitwise AND (&) 5-27
  - bitwise complement (~) 5-16
  - bitwise-exclusive OR (^) 5-27
  - bitwise-inclusive OR (|) 5-27
  - complement 5-16
  - compound assignment 5-36
  - conditional (? :) 5-30

Operators *(continued)*
  - conversions 5-50
  - decrement (--) 5-34
  - differences from Kernighan and Ritchie A-5
  - division (/) 5-21
  - equality (==) 5-25
  - expressions, used in 5-9
  - increment (++) 5-34
  - indirection (*) 5-17
  - inequality (!=) 5-25
  - left-shift (<<) 5-24
  - listed 2-7, B-6
  - logical
    - described 5-28
    - evaluation, order of 5-28
  - logical AND (&&) 5-28
  - logical NOT (!) 5-16
  - logical OR (||) 5-28
  - multiplication (*) 5-21
  - one's complement (~) 5-16
  - precedence 5-37
  - preprocessor
    - differences from Kernighan and Ritchie A-5
    - stringizing A-5
    - token pasting A-5
  - preprocessor specific, listed 8-3
  - relational (>,<,<=,>=) 5-25
  - remainder (%) 5-21
  - right shift (>>) 5-24
  - sequence points, used as 5-12
  - sequential evaluation (,) 5-29
  - shift (<< and >>) 5-24
  - simple assignment (=) 5-35
  - sizeof 5-19
  - subtraction (-) 5-23
  - ternary (?:) 5-14
  - ternary (? :) 5-30
  - two's complement (-) 5-16
  - unary 2-7, 5-14
- OR operators
  - bitwise exclusive (^) 5-27
  - bitwise inclusive (|) 5-27
  - logical (||) 5-28
- Overview 1-1

P

Parameter list 4-36
Parameters
  argc 3-7
  argv 3-7
  envp 3-7
  formal *See* Formal parameters
  macro 8-4
  main function 3-7
Parentheses
  complex declarators, used in 4-10
  expressions, used in 5-10
  function calls, used in 5-3
  function declarators, used in 4-9, 4-33
  macros, used in 8-9
pascal keyword 2-19, 4-14, A-1
Passing by
  reference 7-19
  value 7-16, 7-19
Pointer
  modifier (*) 4-9, 4-28
  void (void *) 4-29
Pointer data-type category 4-7
Pointers
  adding 5-23
  arithmetic 5-23
  comparisons 5-26
  const, modified by 4-28
  conversion 5-48
  declarations 4-9, 4-28, 7-14
  differences from Kernighan and Ritchie A-2
  expressions 5-2
  function calls through 7-17
  functions 7-14, 7-17
  identifiers 5-2
  implicit conversion 5-49
  initialization 4-49
  storage 4-29
  structure 4-28
  subtraction 5-23
  union 4-29
  volatile, modified by 4-28
Portability 4-6
Pound sign (#) *See* Number sign
Pragmas
  described 3-2, 8-2
  differences from Kernighan and Ritchie A-5
  form 8-21
Precedence
  modifiers 4-10
  operators 5-37
Predefined identifiers 8-19
Preprocessor directives *See* Directives
Preprocessor operators
  described 8-1
  listed 8-3
Program execution 3-7
Program structure 3-1, 3-2
Prototypes, function 4-36, 7-1
Punctuation characters 2-4

Q

Quotation marks
  #include directives, used in 8-12
  notational conventions 1-7
  representation 2-5, A-1

R

Recursion 7-23
Reference, passing by 7-19
References to global variables 4-41, 4-42, 4-45
Referencing declarations 4-41
register storage class
  described 4-44
  initialization 4-48
  lifetime 4-40
  local variables 4-44
Relational operators (>,<,<=,>=) 5-25
Representable character set 2-2
Representation, internal 4-6, 4-7
Reserved words *See* Keywords
Restricted constant expressions 5-11, 8-15
return statement 6-16
Return type
  declaration 7-13
  default 4-33
  described 4-34, 7-5
  implicit 7-13
Return value 6-16, 7-12
Right-shift operator (>>) 5-24

S

- _saveregs keyword 4-14
- Scalar data-type category 4-7
- Selection statements 6-13, 6-18
- Sensitivity, case 2-3
- Separators B-6
- Sequence points
  - described 5-1, 5-12
  - listed 5-12
  - operators, other than 5-12
- Sequential-evaluation operator (,) 5-29
- Shift operators (<< and >>) 5-24
- short type
  - conversion 5-42
  - described 4-2
  - differences from Kernighan and Ritchie A-2
  - range of values 4-4
  - storage 4-4
- Side effects
  - expressions 5-1, 5-12
  - macros, used with 8-5, 8-9
  - sequence points, used with 5-12
- Sign extension 2-11, 2-14
- signed
  - char type 4-2, A-3
  - int type 4-2
  - keyword 4-3, A-2
  - long int type A-3
  - long type 4-2, A-3
  - short int type 4-2, A-3
  - short type 4-2, A-3
  - type 4-2, A-2
- Simple variable declarations 4-18
- Simple-assignment operator (=) 5-35
- Single-quotation-mark escape sequence (') *See* Escape sequences
- sizeof operator 5-19
- Source files 3-5
- Special characters 2-4
- Special keywords
  - conversions 7-20
  - declarators, used with 4-29
  - differences from Kernighan and Ritchie A-1
- Standard directories 8-12
- Statement labels
  - described 6-1
  - form 6-11
  - naming class 3-14
- Statements
  - body 6-1
  - break 6-3
  - compound 6-4

Statements *(continued)*
  - continue 6-5
  - do 6-6
  - expression 6-7
  - for 6-9
  - form 6-1
  - goto 6-11
  - if 6-13
  - keywords 6-1
  - labeled 6-1, 6-11
  - listed 6-1
  - null 6-15
  - return 6-16
  - summarized B-14
  - switch 6-18
  - while 6-21
- static storage class
  - described 4-40
  - function
    - declarations 4-46, 7-13
    - definitions 7-4
  - global variables 4-41
  - initialization 4-48
  - local variables 4-44
- Storage
  - bitfields 4-23
  - global 4-40
  - local 4-40
  - type
    - char 4-4
    - double 4-4
    - float 4-4
    - int 4-4, 4-5
    - long 4-4
    - unsigned char 4-4
    - unsigned int 4-4, 4-5
    - unsigned long 4-4
    - void 4-4
  - types
    - array 4-27, 5-6
    - enumeration 4-4, 4-19
    - pointer 4-29
    - structure 4-23
    - union 4-25
- Storage allocation for variables 3-3
- Storage classes
  - described 4-40
  - formal parameters 7-9
  - function
    - declarations 7-13
  - function declarations 4-47
  - function definitions 7-4
  - global variable declarations 4-42
  - local variable declarations 4-44

Storage-class specifiers
  auto 4-40, 4-44
  extern *See* extern storage class
  listed 4-40
  register 4-40, 4-44
  static *See* static storage class
String concatenation 2-15
String literals
  concatenation 2-15
  form 2-14, 5-3
  initializers 4-54
  length 2-16, 5-3
  storage 2-16
  type 2-16
Stringizing preprocessor operator (#)
  described 8-3, 8-5
  differences from Kernighan and Ritchie A-5
Strings *See* String literals
struct type-specifier 4-21
Structures
  conversion 5-49
  declaration 4-21, 4-55
  differences from Kernighan and Ritchie A-3, A-4, A-5
  expressions 5-2
  identifiers 5-2
  initialization 4-48, 4-50
  members *See* Members
  pointers to 4-29
  storage 4-23
  tags
    naming class 3-14
    type declarations 4-55
    variable declarations 4-22
Subscript expressions 5-4, 5-6
Subtraction operator (-) 5-23
switch statement
  constant expressions, used in 6-18
  described 6-18
  differences from Kernighan and Ritchie A-4
  termination of execution 6-3
Symbolic constants *See* Manifest constants
Syntax
  conventions *See* Notational conventions
  summary B-1
System-dependent keywords 2-19

T

Tab escape sequences 2-5
Tags
  enumeration 4-19, 4-55
  naming class 3-14
  structure 4-22, 4-55
  union 4-55
Ternary expressions 5-9
Ternary operator (?:) 5-14
Ternary operator (? :) 5-30
Token-pasting preprocessor operator (##)
  described 8-3, 8-6
  differences from Kernighan and Ritchie A-5
Tokens 2-7, 2-22, B-1
Transfer statements
  break 6-3
  continue 6-5
  goto 6-11
  labeled statements 6-11
Two's complement operator (-) 5-16
Type
  checking *See* Arguments
  declarations 4-55
  modifiers
    differences from Kernighan and Ritchie A-3
  names
    argument-type lists, used in 4-36
    described 4-58
    sizeof, used with 5-19
    void 7-20
  specifiers
    abbreviations 4-3
    const 4-3
    differences from Kernighan and Ritchie A-2
    enum 4-2, 4-19
    fundamental types 4-2
    struct 4-21
    union 4-25
    volatile 4-3
Type-cast conversions 5-50
Type-cast expressions
  constraints 5-10
  defined 5-10
  void, to and from 5-10
typedef
  declarations 4-55, 4-56
  types 3-14, 4-56
Types
  array
    declaration 4-9, 4-26

Types *(continued)*
  array *(continued)*
    initialization 4-48, 4-50, 4-54
    multidimensional 4-27
    storage 4-27, 5-6
  char *See* char type
  const
    described 4-3
    pointers, used with 4-29
  conversions *See* Conversions
  differences from Kernighan and Ritchie A-2
  double 4-2, 4-4, 4-6
  enumeration *See* Enumeration types
  float *See* float type
  floating point
    described 4-2
    internal representation 4-6
  function *See* Return type
  fundamental
    declaration 4-18
    described 4-2
    differences from Kernighan and Ritchie A-2
    initialization 4-49
    listed 4-2
    range of values 4-4
    storage 4-4
  int *See* int type
  integral
    conversion 5-41, 5-44, 5-49
    described 4-2
  long double, differences from Kernighan and Ritchie A-2
  long float A-2
  long *See* long type
  pointer
    conversion 5-48
    declaration 4-9, 4-28
    implicit conversion 5-49
    initialization 4-49
    storage 4-29
  short *See* short type
  signed
    char 4-2, A-3
    int 4-2
    long 4-2
    short 4-2
  structure
    conversion 5-49
    declaration 4-21, 4-55
    initialization 4-48, 4-50
    pointers to 4-29
    storage 4-23
  typedef 3-14, 4-56

Types *(continued)*
  union
    conversion 5-49
    declaration 4-25, 4-55
    initialization 4-48, 4-50
    pointers to 4-29
    storage 4-25
  unsigned char *See* unsigned char type
  unsigned int *See* unsigned int type
  unsigned long *See* unsigned long type
  unsigned short *See* unsigned short type
  user defined 4-55, 4-56
  void 4-3, 4-4
  volatile
    described 4-3
    pointers, used with 4-29

U

Unary expressions 5-9
Unary operators, table 2-7, 5-14
Unary plus operator (+) 5-16
#undef directive 8-11
Underscore character (_) 2-3, 2-17
Union declarations
  types 4-55
  variables 4-25
union type specifier 4-25
Unions
  conversion 5-49
  declaration 4-25, 4-55
  differences from Kernighan and Ritchie A-4, A-5
  expressions 5-2
  identifiers 5-2
  initialization 4-48, 4-50
  members
    described 4-25
    naming class 3-14
    referring to 5-7
  pointers to 4-29
  storage 4-25
  tags 3-14, 4-55
unsigned
  char type
    conversion 5-44
    described 4-2
    differences from Kernighan and Ritchie A-2, A-3
    range of values 4-4
    storage 4-4
  int type

unsigned *(continued)*
  int type *(continued)*
    conversion 5-46
    described 4-2
    portability 4-6
    range of values 4-4, 4-5
    storage 4-4
  keyword 4-3, A-2
  long int type *See* unsigned long type
  long type
    conversion 5-45
    described 4-2
    differences from Kernighan and Ritchie A-2, A-3
    range of values 4-4
    storage 4-4
  short int type *See* unsigned short type
  short type
    conversion 5-44
    described 4-2
    differences from Kernighan and Ritchie A-3
    range of values 4-4
    storage 4-4
  type 4-2, A-2
Unspecified type, pointer to (void *) 4-29
User-defined types *See* Types
Usual arithmetic conversions 5-15, A-3

V

Values
  range of 4-4, 4-5, 4-7
Values, passing by 7-16, 7-19
Variable names *See* Identifiers
Variables
  array
    declaration 4-26
    initialization 4-50, 4-54
    storage 4-27
  auto 4-40, 4-44, 4-48
  communal 4-42
  declarations
    array 4-9, 4-26, 4-27
    described 3-2
    enumeration 4-19
    external 4-40
    form 4-17
    fundamental types 4-18
    global 4-41, 4-42, 4-44
    internal 4-40
    local 4-44
    multidimensional arrays 4-27
    pointer 4-28
    simple 4-18
    structure 4-21
    summarized B-9
    union 4-25
    visibility 4-41
  definitions
    described 3-3, 4-41
    summarized B-15
    visibility 4-41, 4-44
  enumeration 4-19
  extern 4-41, 4-45
  fundamental types 4-18, 4-49
  global 3-10, 4-41, 4-45
  lifetime
    global 3-8, 4-40, 4-48
    local 3-10, 7-12
  local 3-10, 7-12
  multidimensional arrays 4-27, 5-6
  naming class 3-13, A-4
  pointer 4-28, 4-29, 4-49
  register 4-44, 4-48
  simple 4-18
  static 4-41, 4-45, 4-48
  storage allocation 3-3
  structure 4-21, 4-23, 4-50
  union 4-25, 4-50
  visibility 4-41
Vertical-tab escape sequence (\v) 2-5, A-1
Visibility
  described 3-8
  function declarations 4-46, 7-13
  function definitions 7-4
  global 3-9
  nested 3-10
  variable declarations 4-41
  variable definitions 4-41, 4-44
void
  argument-type list 4-34, 4-36
  formal parameter list, used in A-2
  function-return type 4-34
  keyword A-1
  pointer modifier, used as A-2
  pointer to 4-29
  type name 7-20
void type
  conversion 5-49
  described 4-2, 4-3
  range of values 4-4
  storage 4-4
  type specifier A-2

volatile
  keyword A-1
  pointer modifier, used as 4-28
  type specifier 4-3

W

while statement
  described 6-21
  execution, continuation of 6-5
  execution, termination of 6-3
White-space characters 2-3, 2-5

514-210-930
23338